APPROVED BY DRAGON

Anatomy of the DRAGON

by Mike James

Sigma Technical Press

ANATOMY OF THE DRAGON
advanced BASIC programming

Mike James

Copyright © 1983 by M. James

All Rights Reserved

No part of this book may be reproduced or transmitted by any means without the prior permission of the publisher. The only exceptions are for the purposes of review, or as provided for by the Copyright (Photocopying) Act or in order to enter the programs herein onto a computer for the sole use of the purchaser of this book.

ISBN 0-905104-35-8

Published by:
SIGMA TECHNICAL PRESS
5 Alton Road
Wilmslow
Cheshire
UK

Distributors:

Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester,
Wester Sussex, England.

Australia, New Zealand, South-East Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
GPO Box 859, Brisbane,
Queensland 40001, Australia

Typeset, Printed and Bound by Commercial Colour Press, London E7.

Preface

The Dragon can be a frustrating machine to use unless you understand fairly clearly what its BASIC commands do and how they are affected by the way its hardware works. When I first met the Dragon I was very impressed and excited because it was the first low cost computer to use the remarkably powerful 6809 microprocessor. However my initial enthusiasm was a little dampened after trying to write a few programs. Things just didn't always work out as well as I expected and I just couldn't seem to produce the graphics effects that I wanted.

If this story sounds at all familiar to you then do not despair because it has a happy ending. Rather than blame the machine I decided to examine the Dragon and Dragon BASIC to find out what I was doing wrong. This book contains most of the more important things that I discovered about the way the Dragon works. In the course of experimentation and discovery I can honestly say that my initial enthusiasm has been rekindled. Indeed my respect for the Dragon has only been increased by a closer examination. The Dragon is a machine with hidden depths!

If you are looking for a 'cookbook' of Dragon hints and tips that you can simply apply without understanding, then, for the most part, this book isn't for you. For while there are many 'do it this way' sections, its main concern is to build up a picture of the working Dragon so that you will never be surprised by the results of a BASIC command and will find new ways to use your machine.

The Dragon is an interesting computer and a great deal of fun. hope that you will enjoy its fascination as much as I do.

Mike James

Contents

1.	Learning About the Dragon	1
	What You Should Already Know	2
	The Order of Things	3
	What is to Come	3
2.	Inside the Dragon	5
	The Dragon as a Computer	6
	The TV Display	8
	The PIAs - Input/Output	12
	A Complete Block Diagram of the Dragon	14
	The Memory Map	16
3.	Text and Low Resolution Graphics	21
	The Display Modes	21
	The 6847 Video Generator	22
	Setting SAM	23
	Text and Inverted Text - POKEing the Screen	24
	Low Resolution Graphics	29
	Using Low Resolution Graphics	30
	Using Graphics Characters	33
	Using the Other Semi-Graphics Modes	34
	When to use Semi-Graphics	39
4.	High Resolution Graphics	40
	Free Graphics but no Alpha!	40
	Using High Resolution Graphics	40
	The Dragon's Graphics Modes	45
	Using the Graphic Modes	46
	The best of all possible modes	49

5.	Sound	50
	Using SOUND and PLAY	51
	The Sound Generator - a 6 bit D to A	53
	Selecting the Source	57
	Single Bit Sound and Sound Sense	60
	Using AUDIO and MOTOR	61
	BASIC Sounds	63
6.	Advanced Graphics	64
	Paged Graphics	64
	The DRAW Command	70
	GET and PUT: User-Defined Characters	70
	Joysticks and Lightpens	83
	What the Dragon Lacks	85
7.	Interfacing	86
	The 6821 PIA	86
	The Dragon's PIAs	90
	Setting PIAs - the use of AND, OR and NOT	91
	The Keyboard	95
	The Printer Interface - a User Part	99
	The Joystick Interface - an A to D convertor	101
	The Cassette Interface	105
	The BASIC Timer	106
	The Expansion Port	107
	Software vs Hardware - using the Dragon as a VDU	109
8.	Inside BASIC	112
	BASIC's Use of Memory	112
	Dragon Data Formats	116
	The Format of BASIC Lines	118
	Recovering Programs	118
	The TAB Function	119
	Adding Commands to BASIC	120
	Some Useful Memory Locations	121
9.	Introduction to 6809 Assembler and Machine Code	123
	Slow BASIC	124
	The Characteristics of Machine Code	125
	The 6809	125
	The LDA A and ADD A Instructions	126

A Short, Example Program	127
A Second Example- Reversing the Screen	130
Appendix I 6847, 6821 and 6883 Pin Connections	133
Appendix II The Graphics Modes	138

Chapter One

Learning About the Dragon

Using a personal computer is not just a matter of knowing how to write BASIC. However, there is the problem of what to learn next once you have mastered BASIC and produced a few simple programs. You could read some computing theory but in computing it is often difficult to see how theory relates to practice. It is also very often the case that theory is dull and uninteresting until you need to use it. But if you ignore theory until you need it then it is often too late! This book isn't about theory, however, it is about a particular computer - the Dragon - and how it works. It explores and explains the Dragon using essentially nothing more than BASIC.

Rather than starting off from the first principles of computing - binary numbers, boolean logic and so on - this book shows how the complicated hardware and software within the Dragon can be used in ways that are not simply BASIC while at the same time staying within BASIC. The reason for this emphasis on BASIC is two-fold. Firstly, the Dragon is a BASIC machine. In other words, unless you do something extra to it, it runs BASIC when it is first switched on. Secondly, it is possible to discuss many quite complicated ideas from computing theory and practice using the language that most programmers learn as their first language.

If you have recently learned BASIC then you may have been told that to go on to more advanced areas of computing you need to learn another language - machine code or assembler. While this is true to a certain extent, moving on from BASIC to assembler or any other language doesn't increase your understanding of computing - it merely adds yet another language to your list and if you weren't sure what to do with the first one you won't be any happier with the second. Computer languages are like theory, they are a lot easier to learn when you see a need for them! As you progress through this book there will be many occasions when you will realise how to write a program for some

particular application. You will be able to see how to do most of it in BASIC and by seeing what BASIC cannot do for you, you will recognise where its limits lie and understand the need for assembler. In the same way you should be encouraged to learn a little computing theory as you see how useful it can be!

What you should already know

To get the most out of this book it is important that you have come to terms with Dragon BASIC. You do not have to be an expert but you do have to understand how to write straightforward programs of your own and know how to use the Dragon manual to find out how unfamiliar commands work. You won't find any extra explanations of the simpler BASIC commands such GOTO or IF in this book, but you will find examples of how to use the more complicated commands and clarification of their meanings. If while reading this book you find any BASIC that you do not recognise which is used without an explanation, do not give up, look it up in the manual and broaden your horizons!

Although much of this book deals with nothing but software there are many occasions when hardware considerations are important. In most cases you should be able to understand the hardware descriptions without any specialised knowledge - the digital circuitry that makes up a computer is no more difficult to understand than BASIC. However, there are times when some knowledge of simple electronics is needed to get the full value from an explanation or an example. Mostly, this knowledge is nothing more than a rough idea of what a voltage etc is but sometimes it has been impossible to avoid becoming a little more technical. In this case there is no need to worry unless the application is something that you are particularly interested in. Once again you shouldn't be disheartened but instead find a good introduction to electronics. You find that once you actually need to use electronics, its really quite an easy topic to understand.

Reading this book you will find that while it answers many of the questions that you may have about the Dragon, it may also raise questions in other areas. For example, although you do not have to know anything about binary numbers you will find that they often play a secondary role in explanations of the way something works. While such explanations are complete in themselves they might succeed in showing you how useful an understanding

of binary numbers is. And knowing how useful something is, is the best reason for learning about it.

The order of things

It is amusing but true that the best way to read most computer manuals is from the back! The reason for this is that most computer manuals constantly make reference to information that will be explained in later sections. It is a characteristic of computers and computing that understanding something new adds to your understanding of what you already know. This makes the sort of explanation that begins with Chapter One and works its way to the final chapter stage-by-stage very difficult to achieve when the subject is computing. This is the reason why computer manuals have to be read and re-read to get the full value from them and so it is with this book. You will find that some of the earlier chapters mean even more to you once you have read later chapters. Because of this, and the way that your understanding of the Dragon and computing in general will develop, that this is intended to be a book for reading, study and reference. If you are reading a section and find that you do not understand it fully, don't give up and go back to the beginning, carry on reading to the end of the section and *then* re-read it if necessary. You will often find that by the end of a section your initial uncertainty will have been cleared up by some other part of the explanation or via an example. This is advice that I would urge you to follow not only when reading this book but when reading anything to do with computers.

What is to come

A brief outline of the book will help you to see the connections between the various chapters.

Our exploration of the Dragon begins with an outline of its hardware in Chapter 2. The purpose of this chapter is to construct a block diagram of the machine and a general memory map, both of which are built on and refined in later chapters.

Chapter 3 introduces many of the ideas concerned with the Dragon's graphics display by way of explaining low resolution graphics and text. Some extra graphics modes that offer a fairly high resolution in nine colours are also described along with practical examples.

Chapter 4 continues where Chapter 3 left off in pursuit of an understanding of all of the Dragon's display modes. The standard high resolution graphics modes are explained and two new ones are introduced. A general discussion of how to use Dragon graphics brings the chapter to a close.

Chapter 5 explains the Dragon's sound generator. The hardware is described along with the BASIC commands SOUND and PLAY. The use of the cassette recorder as a sound source is also discussed and a method for synchronising external sounds is described.

Chapter 6 is once again about the Dragon's video display. This time the subject is the main commands that Dragon BASIC provides for graphics and how they can be used to good effect. Paged animation is described as a way of making things move smoothly and quickly using nothing but BASIC. The DRAW command is explored, problems identified and a way of creating user-defined characters is explained. GET and PUT are explained in detail including the internal formats used to store graphics information in arrays.

Chapter 7 is about interfacing and is the most hardware oriented of all the chapters. The use of PIAs is explained along with methods of using BASIC for 'bit manipulation'. The major practical topics of this chapter include - the printer port, the keyboard, the joystick interfaces and the timer. Applications include a repeat key for BASIC programs, a limited user port and connecting things to the A to D convertor.

Chapter 8 is about Dragon BASIC and is the most software oriented chapter. Details of the internal formats used by Dragon BASIC to store data and programs, are given along with an explanation of how memory is used and allocated. A variable dump program and a procedure for recovering lost programs are given and the TAB function is explained.

Chapter 9 is the final chapter but rather than being an ending it takes us on to new territory in the form of 6809 assembly language. On many occasions earlier in the book the need for assembly language has limited what has been possible. The purpose of the final chapter is to explain what assembly language is and to give an idea of the 'flavour' of assembly language programming.

Chapter Two

Inside the Dragon

The Dragon is a small computer with a lot of interesting features. A knowledge of its overall design will help you not only to use these features to good effect but also to understand the reasons behind most of the things that the manual tells you to do. Once you see that there are reasons *why* things work then you will often find new and better ways of using the Dragon. The only trouble is that an understanding of the overall design involves studying the 'hardware'. Now if you have been a 'software only' person until this time then tackling something more than BASIC may worry you. However, if you have mastered BASIC then you should be perfectly able to understand the hardware descriptions in this chapter.

The Dragon is different from many other personal computers in that it uses an advanced and very powerful microprocessor - the 6809 - designed and manufactured by Motorola. This isn't the only advanced product from Motorola that has found its way into the Dragon. In particular, the graphics - text, low and high resolution - are produced by a 6847 video generator, the sound, cassette, printer and joystick interfaces are provided by two 6822 Peripheral Interface Adaptors and the overall working of the separate parts of the Dragon are synchronised by a 6883 Synchronous Address Multiplexer (SAM). It is true to say that the internal workings of the Dragon owe so much to Motorola that if Motorola didn't exist neither would the Dragon!

Much of the detail of the operation of Dragon is described in specific chapters in the rest of this book. For example the video generator is described in the chapters on graphics and the way that a 6822 PIA is used to produce sound effects is described in the chapter devoted to Dragon sounds. As these finer details are missing from the picture of the Dragon given in this chapter you shouldn't be too worried if you feel that you don't understand everything. The purpose of this chapter is to start the ball rolling with a general description

of the machine. The whole picture is something that you will only appreciate when you have read all the chapters in this book and used some of the information they contain. The main aim of this chapter is to build up a block diagram of the Dragon's hardware that can be used in later chapters to see how the more specialised descriptions fit into the entire machine. As a spin off from the process of building up a block diagram, we will also construct a memory map of the Dragon showing where all the areas of interest are located and roughly what each is used for.

The Dragon as a computer

There are certain features that all computers have in common and the Dragon is no exception. The basic anatomy of any computer can be seen in Fig 2.1. The Central Processing Unit (CPU) is usually considered to be the

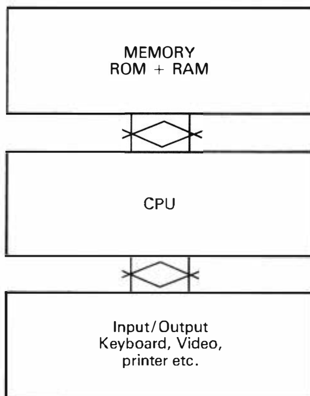


Fig 2.1 The parts of a Computer

most important and interesting part of a computer in that it is responsible for carrying out the instructions contained in any program. As you might guess, in this role the CPU mainly affects the speed that your programs run at and unless you want to get involved in assembly language programming, this is the only way that the CPU makes itself felt to the average user. (For more information on assembly language, see Chapter 10.) However, it is comforting to know that the CPU inside the Dragon is a very fast and sophisticated device - the Motorola 6809. If you are contemplating learning or using assembly language then the 6809 is one of the best to use and, on the other hand, if you are only interested in running BASIC programs then the 6809 will not let you down from the point of view of speed.

The second major part of any computer is its memory. Memory comes in two distinct types, RAM and ROM. ROM (Read Only Memory) is used to hold information that never changes. Within the Dragon its main function is to hold the program that implements the rules of BASIC - that is the BASIC interpreter. This is stored in two 8K ROMs making a total of 16K of ROM storage. The uses that RAM (Random Access Memory) is put to are very much more varied. Within the Dragon RAM is used to hold the text of a BASIC program, any variables, strings and arrays that the BASIC program may use and a large amount of RAM can be used up to store the information used to produce the video display. The Dragon comes equipped with 32K of RAM and although this sounds like a lot it can be used up by some graphics modes and large arrays. Physically the RAM is supplied in the form of eight 4864 dynamic RAM chips. To a certain extent this information is of academic interest only because, although the Dragon has been designed to take smaller 16K chips as well as the 32K chips it is supplied with, it is difficult to think of a reason for wanting to swap them.

A point of interest is that that the Dragon uses 'dynamic' RAM chips. In fact RAM chips come in two different types - static and dynamic. Static RAM chips are the simplest to use in a computer because they retain whatever information is stored in them as long as the power is supplied without any extra attention. However, static memory chips are expensive and not available in very large capacities. Dynamic RAM chips on the other hand are more trouble to use but they are cheap and 32K of memory can be supplied in just eight chips. The reason why dynamic memories are so much extra trouble is that unlike static RAMs they have to be kept continually active to retain the information stored in them. Put simply this activity takes the form of continually reading every memory location - this is called 'refreshing' the memory - and it is the need for refreshing that complicates the use of dynamic RAM. In

most computer systems the use of dynamic RAM involves a whole collection of extra chips to look after refreshing. In the Dragon this operation is taken care of by a single chip, the SAM (Synchronous Address Multiplexer), more of which later.

The CPU, any other device using the memory, and both RAM and ROM need to be connected to two 'system buses' - the address bus and the data bus. (A 'bus' is simply a group of connections that can be though of as serving one purpose within the computer.) The address bus is used to select one of the many memory locations numbered from location 0 to location 65535. If you are familiar with binary numbers you will know that this range of numbers needs 16 bits to represent it and so the address bus has to have 16 different connections, one for each bit. The data bus is different from the address bus in that information can pass in both directions, from the memory during a read operation and to the memory during a write operation. For this reason the data bus is said to be 'bi-directional'. Each memory location can hold a number in the range 0 to 255 and this corresponds to a binary number with eight bits, or one byte. Thus to transfer data to and from the memory requires a data bus with eight different connections, one for each bit. These two system buses thread their way through the Dragon connecting any two components that either need to received an address or need to send or receive data. There is in fact a third bus that threads its way though the Dragon along with the data and address bus - the control bus. This is a collection of connections that do a wide range of different jobs concerned with controlling the operation of the machine. For example, one of the connections is used to decide if a memory operation will be a read or a write. The control bus is something that is better described as and where it is crops up.

The third part that all computer have in some for of another is input/output. In the Dragon the main input/output is via a keyboard and standard TV screen. As well as these two major methods there are a number of other more specialised ways that information is passed in and out of the Dragon - the joystick interfaces, the cassette interface, the sound channel and the printer interface. The only one of the input/output devices to affect the overall working of the Dragon is the TV display and this deserves a section all to itself.

The TV display

To understand the workings of the Dragon's TV display we first have briefly to examine the way that a typical computer produces its TV display - just to be in a position to appreciate how clever the Dragon is in doing it better! Without

going into too much detail, the elements of a standard video system can be seen in Fig 2.2. This method of producing a video display is called 'memory

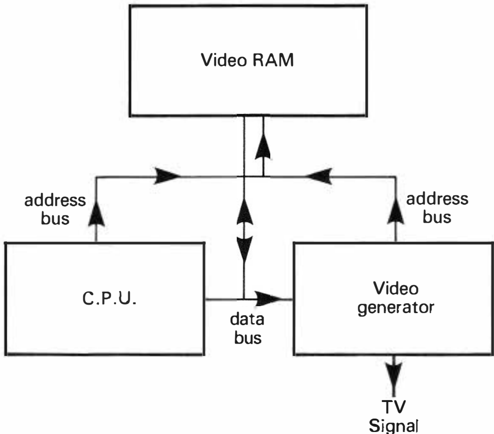


Fig 2.2 A Standard Video System

mapping'. The principal idea is that there is a special area of RAM, the 'video RAM', that is shared by the CPU and by another device, the 'video generator'. In early systems the video generator was the name given to a whole collection of chips which were necessary to produce any sort of TV picture. However, these days all of the functions originally carried out by this collection of chips can be carried out by a single device. The video generator has two things to do, firstly it must produce all the timing signals that a TV set needs to display a steady picture and secondly it must read the video RAM and use the information contained in it to display the correct patterns on the screen. The generation of the timing signals causes no real problems. All that

is required is a pulse marking the beginning of each TV frame - the 'frame pulse' or 'frame sync' - and a pulse to mark the beginning of each scan line - the 'line pulse' or 'line sync' - see Fig 2.3. The real problem lies in the need for the video generator to access the video RAM for information about the screen display. As this portion of RAM is also used by the CPU to store, and hence alter, what is being displayed on the TV you might be able to guess that the trouble lies in deciding which of the two devices, CPU or video generator, is using it. This synchronisation problem is often ignored by the designers of personal computers with the result that the display tends to show random 'sparkles' as the CPU and the video generator compete for the memory.

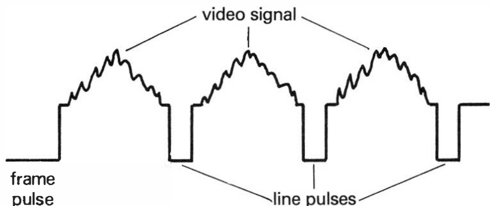


Fig 2.3 TV Signal

The solution adopted by the Dragon is to use a video generator chip, the MC6847, in conjunction with the SAM chip already mentioned. The SAM chip not only refreshes the dynamic RAM as described in the last section, it also determines which of the CPU and video display chip can gain access to the memory. In fact the SAM chip is the only device in the Dragon that is allowed to control the memory. When the CPU needs to access the memory the SAM chip is responsible for taking the address from the CPU and either storing or retrieving the data. The arrangement of the three most important chips in the Dragon, the 6809 CPU, the 6847 video generator and the 6883 SAM can be seen in Fig 2.4. There is only one problem. As there is no address bus connecting the video display chip with the memory, how does it manage to retrieve data at the correct time and in the correct order. The answer is somewhat surprising. The SAM chip mimics the operation of the video generator and automatically produces the necessary data to the video generator. At first this may sound like a very complicated and round about way of doing things. Its advantages are that the Dragon can be made quite a

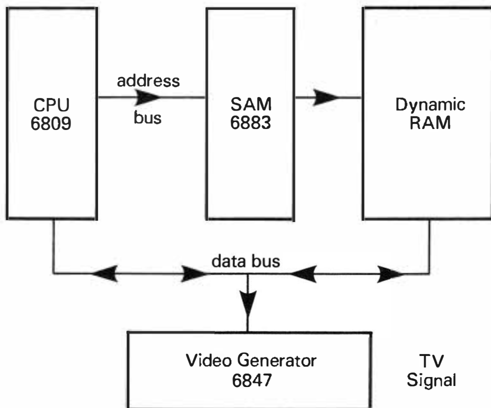


Fig 2.4 The 6809, 6883 & 8647 as a video system

lot simpler in terms of its construction by not having to run 16 address lines between the memory and the video generator and the SAM chip can make sure that the CPU and the video generator never try to use the memory at the same time. The crudest way of keeping the video generator and the CPU out of each other's way is to stop the CPU from doing anything during the time that it takes to display a TV picture. If your realise how much this would slow the Dragon down you will be relieved to hear that this is *NOT* how the SAM chip solves the problem! Instead what happens is that the CPU and the video generator are each given an opportunity to access memory alternately. This scheme is particularly economical because the time that the video generator is given couldn't be used by the CPU anyway. This interleaving of the CPU and video generator is entirely the responsibility of the SAM chip, see Fig 2.5, and is one of the reasons that the Dragon's video display is flexible without losing any of the CPU's innate speed.

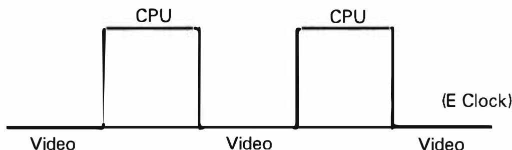


Fig 2.5 The interleaving of memory access

One of the disadvantages of using the SAM chip doesn't really show itself if you just use the Dragon according to the manual. As mentioned earlier, to make sure that the video generator chip gets the data it needs at the time that it needs it the SAM chip has to mimic its functioning. What this means is that if you instruct the video generator chip to change to a different graphics mode you have to remember to tell the SAM chip about it. This is normally taken care of by the Dragon's BASIC interpreter and so there is no need to worry about this complication. However, there are graphics modes that can be produced by the 6847 chip that Dragon BASIC doesn't mention and some of these modes are very useful! To use these extra modes it is obvious that you have to set the video generator but what might have escaped your notice is that the SAM chip also needs to be set to the same mode. The details of the SAM chip and the video display chip can be found in Appendix I but it is better to wait until they are needed in Chapter 3 before looking at them.

The PIAs - input/output

The Dragon is very definitely a 'big chip' machine! We have already seen how both the video generator and the SAM are large chips that replace the collections of small chips found in other machines. There are two other large chips in the machine and both of them are 6821 PIAs from Motorola. PIA stands for 'Peripheral Interface Adaptor' but this doesn't really convey what the chip does. A PIA is in fact a general purpose input/output device and each PIA within the Dragon provides 20 connections that, subject to some restrictions, can be used either as inputs or outputs. A PIA is usually thought of as two groups of connections called the A side and the B side. The A side

consists of a group of eight 'data lines' usually named PA0 to PA7 and two control lines called CA1 and CA2. The data lines can act as either inputs of outputs and can be treated by a programmer rather like a normal memory location. That is, there is a memory location within the Dragon that corresponds to the A side of the PIA and writing to it alters the output lines and reading from it returns the current state of the input lines. If this seems a little complicated it will be easier to understand after some practical examples in Chapters 5 and 6. The two control lines CA1 and CA2 are a little more complicated to handle than the data lines and details will be left until later. However, CA1 can be used as an extra input and CA2 can be either an extra input or an extra output. The B side of the PIA is almost identical to the A side and provides eight data lines called PB0 to PB7 and two control lines CB1 and CB2, (see Fig 2.6).

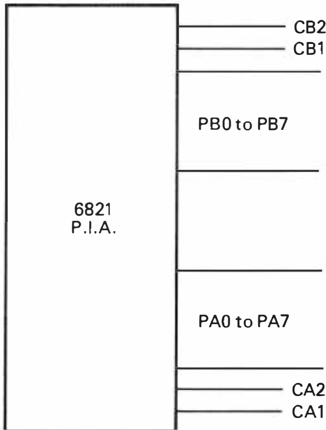


Fig 2.6 The 6821 P.I.A.

PIAs are very useful in that they allow a machine to communicate or interface with the outside world and many personal computers provide spare PIAs for the user to have fun with. These PIAs are often called 'user ports'. You probably already know that the Dragon doesn't have a user port so you might be wondering what it is doing with the two PIAs inside it and how we can go about connecting things to the Dragon. The answer to the first part of the question is that PIA0 looks after the keyboard and provides the parallel printer interface and PIA1 handles the cassette interface, part of the printer interface, the sound output, the joystick inputs and various other things that will be discussed later! Although the two PIAs are fully occupied with 'internal duties' it is possible to make the parallel printer interface do things that it was never intended for!! In other words the printer interface can be pressed into service as a restricted user port. Another good reason for learning a little about the workings of the PIAs is that many of the Dragon's internal facilities that are controlled by the PIAs can be extended and modified. For example, one of the more irritating shortcomings of the Dragon is its lack of a repeat key. Knowing how PIA0 reads the keyboard makes it possible to read the keyboard directly and provide a repeat key. All in all the PIAs are two very important components from the user's point of view. The details of the 6821 PIA can be found in Appendix I along with the SAM and video generator but once again it is better only to look at this appendix as and when it is required in reading later chapters.

A complete block diagram of the Dragon

Now that we have made the acquaintance of the most important components in the Dragon and looked briefly at their inter-relationships it is possible to construct a block diagram of the machine, Fig 2.7. You should be able to identify the chips that we have been discussing and know roughly what the 6809 CPU, the RAM and ROM, the SAM, the 6847 video generator and the two PIAs are doing. You should look very carefully at the data bus and the way that it connects the various components together. Notice the address bus and the way that it is intercepted by the SAM chip - the reason for this was explained earlier. The only device that hasn't been discussed earlier is the modulator. The only function that this device performs is to change the video and sound signals into a modulated UHF signal suitable for reception by a domestic TV set. The details of the I/O devices such as the sound generator and the cassette interface will be dealt with in later chapters. For the moment all you should concentrate on is understanding the overall configuration of the Dragon - the practical details are yet to come!

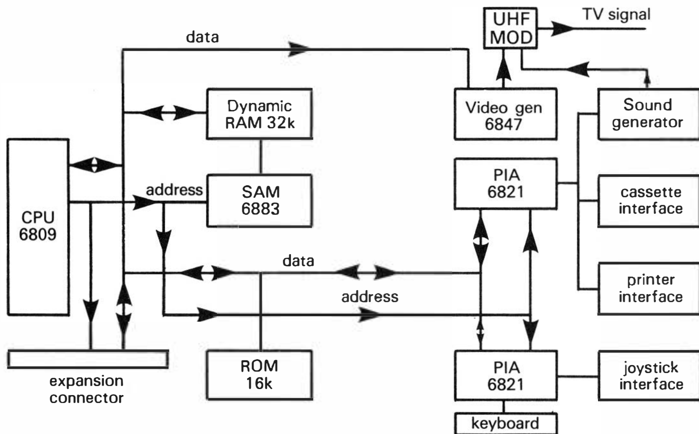


Fig 2.7 A Block diagram of the Dragon

The memory map

Although the system block diagram is important to a certain extent, from the programmer's point of view the system memory map is even more important. For while the block diagram shows you what is IN the machine the memory map tells you WHERE things are. Every device inside the Dragon that is either connected to the address bus or to the SAM chip has a range of addresses that it responds to. If you want to write programs that make use of these devices then it is obviously necessary to know the addresses that they respond to. The only complication is that addresses within computers are normally specified in 'hexadecimal'. If you don't know about hex numbers then don't worry because the only important thing to know is that in hex we count from zero to 15 before incrementing the next digit. As the decimal system only provides ten digits 0 to 9 we have to invent five new ones to get to 15. Conventionally the letters A to F are used. So counting in hex goes -
hex = 0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F,10,11,12...1F,20...
dec = 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18...31,32...
If this seems complicated then take heart because your Dragon will convert any decimal number to hex and any hex number to decimal without any effort. After all what else are computers for! To convert to hex simply use the HEX$ function as in -

```
10 INPUT X
20 PRINT X;"IN DECIMAL = ;"HEX$(X);"IN HEX"
30 GOTO 10
```

If while reading this book or writing a program you want to know what a decimal number is in hex then type PRINT HEX$(x) where x is the decimal number in question in direct mode to your Dragon. The problem of converting hex to decimal is even less troublesome because, although the manual doesn't mention it, your Dragon will accept hex numbers as well as decimal numbers. To signify that a number is hex all you have to do is to write &H in front of it. For example PRINT 100 will print '100' on the screen but PRINT &H100 will print '256' on the screen because &H100 is taken to be a hex number which is converted to decimal before being printed out. To see what the decimal equivalent of any hex number is try -

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

If you type a decimal number into this program then it will print the same number out but if you type a hex number, &HFF for example then it will

convert it to decimal and then print it out. If you want to know the decimal equivalent of any number while you are writing a program then just enter

PRINT &Hx

in direct mode where x is the hex number that you want converted. In general however, there is no need to convert hex to decimal for use in programs because Dragon BASIC will allow you to use a hex number wherever you could use a decimal number. (From now if there is likely to be any doubt, a hex number will be written with &H in front to make sure that it is not mistaken for decimal.)

You might be wondering why hex is used at all. The first reason is that large decimal numbers are difficult to remember and hex numbers use fewer digits than decimal. For example the hex address C000 is 49152 in decimal. The second reason for using hex is that computer addresses tend to be either powers of two or at least multiples of two and such numbers look particularly simple in hex. (There is a third reason for the general use of hex, it is easier to convert to binary, but this need not concern us here.)

The general system memory map can be seen in Fig 2.8. The lower half of the addresses correspond to RAM which is used for a number of different purposes including storing any lines of BASIC that you might write. A more detailed map of RAM use can be seen in Fig 2.9 and this will be discussed in the coming chapters. The BASIC ROMs and the cartridge ROMs take up 24K of memory and there is not much more to be said apart from pointing out that there are some machine code routines within the BASIC ROMs that are occasionally useful. The most interesting area of the Dragon is to be found in the top 8K of the memory map. This corresponds to the I/O device area and is where the SAM, video generator and the PIAs can be found. This area will be explored in detail later but a closer view of this area can be seen in Fig 2.10.

The memory maps presented in this section start us out on the road of exploration - they tell use where things can be found. Once we know where they are the next step is to discover what to do with them.

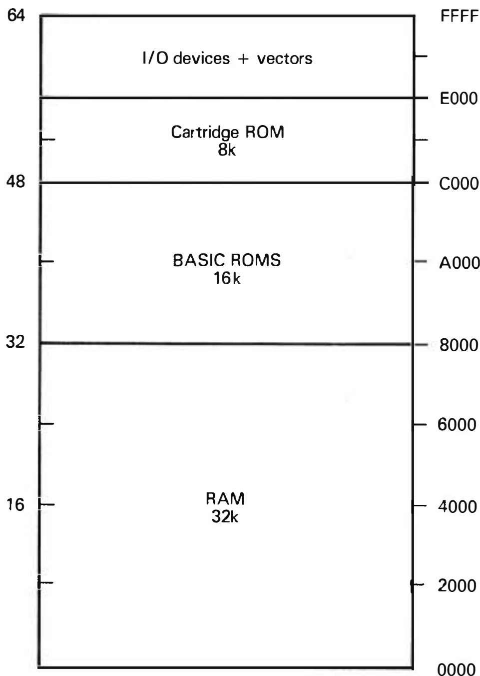


Fig 2.8 General system memory map

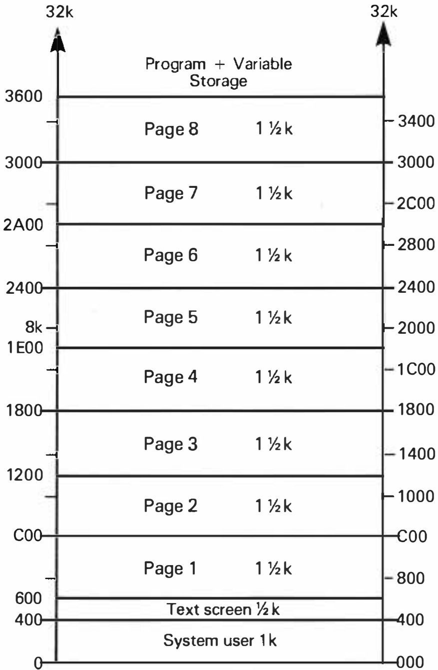


Fig 2.9 RAM use

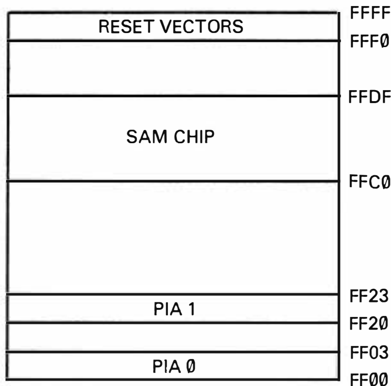


Fig 2.10 Memory Map of the I/O area

Chapter Three

Text and Low Resolution Graphics

In this chapter we look in detail at the way that the 6847 video generator produces the Dragon's range of text and low resolution graphics modes. The reason for looking at this isn't just idle curiosity, although there is much of interest, it is a way of expanding the Dragon's range of graphics modes. It may come as something of a surprise to learn that the Dragon has even more graphics modes than BASIC offers but it's true! A second reason for looking at the way the display works is that it helps to understand some of the unexpected behaviour of the Dragon's low resolution graphics. Some of the details of how things work are a bit technical - after all the Dragon is a complicated machine - so you will probably have to read this chapter a few times before you really grasp everything that it explains. It is certainly worth making the effort to understand how your Dragon works, because by understanding it you stand a better chance of inventing new ways of using it. However, if for the time being you just want to use the techniques described then you can always skip the explanations and look at the summaries within the technical sections and go straight on to the sections that deal with 'using' the features discussed.

The display modes

The Dragon's video generator can work in three distinct ways, giving rise to

- alphanumeric modes
- semi-graphics modes
- full graphics modes

The three modes differ in the way that they interpret the data stored in memory (and sent to the video generator by the SAM chip). In alphanumeric mode each byte of data corresponds to the code of the character to be displayed. The exact dot pattern of the character (i.e. its shape) is obtained from an area of memory inside the video generator.

In the semi-graphics mode each byte of data contains information about whether or not a point on the screen should be 'on' or 'off' and additional information about what colour 'on' points should be. The reason that this is called a 'semi-graphics' mode is that it is not possible to control the colour of every point independently. A byte of information will set a group of points 'on' or 'off' and set all the 'on' points to the same colour. Points that are 'off' are always black and unfortunately this is something that cannot be changed.

The full graphics modes are much easier to use than the semi-graphics modes because each byte of information directly controls the colour of a group of points. In other words, points are not 'on' or 'off', they are simply assigned a colour. This means that you can assign a colour to any point without worrying about affecting the colour of any other point.

These descriptions of the three different modes will make more sense after you have had some practical experience of each one in this and the following chapter. However, before moving on to software it is necessary to look at the way that the 6847 video generator chip can be made to change its display modes.

The 6847 video generator

If you look back at the system block diagram given in Chapter 2 you will see that the video generator is not connected to the address bus. As has already been explained, this is not a problem as far as getting display data to it is concerned because the SAM chip addresses the memory on the video generator's behalf. However, it does cause something of a problem when it comes to giving commands to the 6847 to set its mode of operation. Without being connected to the address bus, how can the video generator know that the CPU is sending data to it in particular. The answer to this problem is that the 6847 chip has a number of input pins that can be used to set its mode of operation. Instead of these pins being permanently wired to give a fixed mode they are connected to output lines of PIA1, one of the two PIAs discussed in

Chapter 2. The control pins of the video generator can be seen in table 3.1 along with other information that will be explained as we go along.

Table 3.1

The 6847's control pins

Pin no.	name	controlled by	function
39	CSS	PIA1 PB3	selects one of the two colour sets
30	GM0	PIA1 PB4	graphics control line 0
29	GM1	PIA1 PB5	graphics control line 1
27	GM2	PIA1 PB6	graphics control line 2
35	A/G	PIA1 PB7	alpha-numeric/graphics select
34	A/S	data b7	alpha-numeric/semi-graphics select
32	INV	data b6	invert alpha-numerics
31	INT /EXT	as GM0	internal/external character ROM

The meaning of this table will become clear as we examine the different display modes of the video generator chip. All that needs to be clear at this time is that these control pins are the only way to give the video generator chip instructions about what mode is required and some of these pins are connected to the B side of PIA1. We will find out how to change the state of the PIA as the need arises later in this chapter.

Setting SAM

Now that we have some idea of how to set the mode of the video generator chip we can go on to find out how to set the SAM chip to the same mode. This is necessary, you will recall, because the SAM chip has to mimic the operation of the video chip to produce the correct data at the correct time. This is indeed the normal operation of the SAM chip but by setting it to one mode and the video chip to another we can create a few completely new graphics modes. It is difficult to describe exactly how mixed mode operation works, but roughly speaking the SAM chip can be made to send more data to the video chip than strictly necessary and so produce a higher resolution.

The SAM chip is connected to the address bus which it uses to control the memory and to accept commands from the CPU. It does this by recognising addresses from FFC0 to FFDF as reffering to itself. That is, any use of an

address in this range will change the way the SAM works. The exact details of the way that the SAM is affected are rather odd and certainly no other device in the Dragon is controlled in quite the same way. The addresses that the SAM uses are best thought of in pairs such as FFC0 and FFC1, FFC2 and FFC3, i.e. an even address and an odd address. Each pair of addresses controls one aspect of the way the SAM works. If you write data to the even address of the pair the particular aspect concerned is turned off and if you write data to the odd address it is turned on. Notice that it doesn't matter what data you write to the address it is simply the fact that you *used* the address that the SAM chip notices. Each pair of addresses can be thought of as a switch with the even address corresponding to 'off' and the odd address corresponding to 'on'. As you probably know, in computing 'off' can be represented by0 and 'on' by 1, This leads to another way of talking about the way the even and odd addresses work - you can say that even addresses 'set' the function, while odd addresses 'clear' them or set them to zero. If you look at Appendix I, you will see a complete list of the address pairs and a brief description of what they do. Within the Dragon, many of these conditions of operation cannot be changed and so they are of little interest to us. From the point of view of the graphics modes there are only three address pairs that are used and they are -

Address pair (clear/set)	**name**
FFC0/FFC1	V0
FFC2/FFC3	V1
FFC4/FFC5	V2

Which mode the SAM will run in depends on which combination of V0 to V2 are set and which are cleared. Details of how to set up a particular mode will be given later.

It is worth summarising what we have learned so far. Firstly , the video generator is controlled by some of the output lines of PIA1. Secondly, the SAM chip's mode is set by writing to odd or even addresses in the range FFC0 to FFC5.

Text and inverted text - POKEing the screen

After so much theory it is time to look at perhaps the simplest Dragon display mode - text and inverted text. Whenever you switch the Dragon on

the video display chip and the SAM chip are set to produce a text display. The BASIC command SCREEN 0,c will also set a text display with either black on green, if c is 0, and black on orange, if c is 1. As text mode is so easy to get into, there is not too much point in explaining in detail how to set the video chip and the SAM to produce it. (It is, however, included in Appendix II.) What is worth looking at is the way the data in memory produces characters on the screen. Generally, the most important things to know are what you have to store in a memory location to produce any given character on the screen and which memory location you have to store it in to make it appear at a given position.

The data for the text screen display is normally stored starting at address &H400 and going up to 5FF. The number stored in each memory location determines what is displayed on the screen within one character location. As there are 32 character locations on 16 lines you can see that this implies that there are 512 memory locations used for the text screen, which is indeed true. Which character is displayed is actually determined by just the first six bits of the contents of the memory and as each location can store a number consisting of eight bits the question is: what are the remaining two bits used for? Bit 6 (notice that this is the seventh bit as we number bits starting from zero) is in fact used to select between two distinct text display modes - inverted and non-inverted. If you look back at table 3.1 you will see there is a pin called INV and that it is controlled by 'data b6'. When bit six is 1 the characters are displayed 'inverted' which corresponds to black on green, i.e. the normal way that characters are displayed. However, if bit six is 0 then a character will be displayed green on black, i.e. the way they are following SHIFT 0. Most computers store ASCII codes in their video memory to determine which character will be displayed. That is, to display a letter on the screen the corresponding memory location would have the letter's ASCII code stored in it. However, the Dragon is a little more complicated than this. So it is worth examining not only the way that each memory location corresponds to a screen location but the way the number stored in it determines which character is displayed on the screen. Try the following program

```
10 INPUT A$
20 FOR I = &H400 TO &H5FF
30 POKE I,ASC(A$)
40 FOR J = 1 TO 100:NEXT J
50 NEXT I
60 GOTO 10
```

This program will POKE the ASCII code of any letter into every location in the screenmemory. Line40 is a delay loop so that you can see what happens. You can learn two things by watching this program. Firstly the letter that appears in every location is not always the same as the letter you type into A$ - for example try '*' or the number 1. Sometimes the characters appear on the screen the opposite way round to the way you typed it, i.e. it appears green on black. If you try entering characters after pressing SHIFT and 0 to get into 'lower' case you will be even more confused. Now when you type in letters, what you see on the screen are numbers or symbols! This apparently confusing display is easy to understand when you remember that, as already described, the Dragon interprets bit 6 of every memory location as an invert/non-invert instruction. The trouble is that ASCII uses bit 6 to distinguish numbers from letters and bit 5 to distinguish upper from lower case characters. You should be able to see that to make all upper case characters, digits and symbols display in inverted mode and all lower case characters display in non-inverted mode, something other than the ASCII code will have to be stored in memory. For example, if you POKE the ASCII code for the letter A to a memory location then the actual number that you store is 65 in decimal or 01000001 in binary. Now if you look at this binary number you will see that bit 6 is a one and this means that the character will be displayed in inverted mode. The remaining six bits are 000001 and these determine which character will be displayed on the screen. In other words storing 000001 or 1 in decimal determines that the letter A will be displayed and depending on the state of bit 6 it will either be inverted or non-inverted. If bit 6 is a 0 then it will be displayed non-inverted and this is what the Dragon displays for a lower case A. If you have followed this complicated description you will see that although the ASCII code for a lower case A is 97 in decimal you have to POKE the screen with a 1 to produce an inverted A and this is a long way from the ASCII code for lower case A.

Whenever you PRINT a character to the screen the Dragon automatically carries out a conversion from the ASCII code that the keyboard produces and that BASIC uses to a display code that is stored in the screen memory. Rather than give a table showing how ASCII can be translated to this display code it is more useful to give a number of rules for conversion -

```
IF ASCII< 64 THEN CODE=ASCII+64
IF ASCII> =64 AND ASCII< =96 THEN CODE=ASCII
IF ASCII> =96 AND ASCII< =128 THEN CODE=ASCII-96
```

Where ASCII is the ASCII code for a character and CODE is the number that has to be POKED to the screen to display it. Notice that not all the ASCII

codes recognised by the Dragon correspond to something printable on the screen - for example the cursor control codes will not appear on the screen. One other interesting point is that there are display codes that correspond to ASCII codes that the keyboard cannot produce. For example how do you display green on black digits. The keyboard doesn't produce ASCII codes for 'lower case digits' because such a thing doesn't make sense! But if you POKE codes between 48 and 57 you will see green on black digits. The reason for this is that if a code produces an inverted character then subtracting 64 from it will produce a non-inverted character even if the keyboard will not produce an ASCII code for it. This is because in this case subtracting 64 is the same as setting bit 6 to zero. (In the example given above the display code for inverted 0 is 112 and 112-64 gives 48.)

All this makes using the Dragon's text screen directly either by POKEing it from BASIC or from machine code a little more difficult than you might expect.

To summarise, the Dragon handles characters in BASIC and from the keyboard in the standard computer code, ASCII. When a character is PRINTed to the text screen the ASCII code is changed so as to make lower case characters display in non-inverted mode (green on black) and upper case characters, digits and other symbols display in inverted mode (black on green).

Now that we have the intricacies of the Dragon's display code sorted out we can turn our attention to the way that each memory location corresponds to each screen position. Looking back at the screen POKE program you should be able to see that the letter fills the screen from left to right and moving down. In fact in the same order as used to number printing positions by the PRINT @ statement. If you want to POKE a character whose display code code is d to column x and row y use -

```
POKE x+32*y+&H400,d
```

At this point it might be a good idea to tackle a small practical problem. The following short program reads characters in from the keyboard using the BASIC command INPUT and then displays them on the screen but without using the BASIC command PRINT.

```
10 X=0
20 Y=0
30 INPUT A$
40 FOR I= 1 TO LEN(A$)
```

```
 50 A=ASC(MID$(A$,I,1))
 60 IF A<64 THEN C=A+64
 70 IF A>=64 AND A<=96 THEN C=A
 80 IF A>=96 AND A<=128 THEN C=A-96
 90 POKE &H400+X+32*Y,C
100 X=X+1
110 IF X>31 THEN Y=Y+1:X=0
120 NEXT I
130 IF X<>0 THEN Y=Y+1
140 GOTO 3
```

This short program is in fact a BASIC version of what happens in machine code when you use the PRINT command. You should be able to follow the way that the printing position is moved on by one after each letter is printed and the way that the ASCII code of each letter is changed to a display code. The program works for all characters both upper and lower case but it doesn't handle control codes such as ENTER and it doesn't scroll the screen once it is full.

At this point you may be wondering how the video generator converts a display code, which certainly doesn't carry any information about the shape of the letter that has to be displayed on the screen. The answer is that there is a chunk of ROM inside the video chip that stores the dot shapes of all of the 64 characters that the Dragon can display. If you go back and look at table 3.1 you will see that there is a control pin called INT/EXT. What this does is to make the video chip use either its internal character definitions or an external RAM or ROM chip. This could obviously be used to give the Dragon upper and lower case characters but the modification would involve unsoldering the video chip and connecting all its unused address lines to an external character generator - this is for expert experts only!

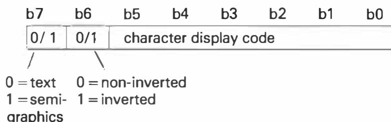
The only question left unanswered is - what the eighth bit in the video data is used for. Once again looking back at table 3.1 shows that bit7 is connected to the A/S pin. The A/S pin changes the display mode from alphanumeric to semi-graphics which forms the subject of the next section. If bit7 is zero then the data is displayed in text mode (inverted or non-inverted according to bit 6) as a character. If bit 7 is a 1 then the data is not translated to a character but is used to determine the colour of four small squares that occupy the same space on the screen that a character would. You should recognise this semi-graphic mode as Dragon low resolution graphics. It is the use of bit 7 that allows text and low resolution graphics to reside on the screen at the

same time - the video chip changes mode as it goes along according to the value of bit7. In fact, if you think about it the Dragon's text screen is a mixture of three distinct video generator modes - alphanumeric inverted, alphanumeric non-inverted and semi-graphics!

Summary

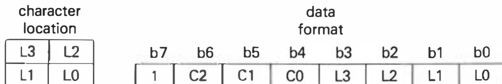
1) The Dragon's screen memory is organised in such a way that if you want a character to appear at column x and line y you should POKE its display code to location x + 32*y + &H400

2) Format of a byte of data used in text mode is-



Low resolution graphics

If bit 7 of the contents of a screen memory location is set, the following seven bits of data are not interpreted as a text code. Instead the next three bits are used to define the colour of the character location and the last four bits determine the state of each of four small squares within the character location - i.e. whether it is 'on' or 'off'. In other words, the formats of the data and of the screen display are-



The bits L0 to L4 determine which of the squares are 'on' and which are 'off'. For example, if L0 is 1 then the square marked L0 is 'on', if L0 is0 then itis 'off' and so on. The colour bits, C2 to C0 determine the colour of each of the squares that are 'on' in the following way

C2	*C1*	*C0*	*n*	*Colour*
0	0	0	0	Green
0	0	1	1	Yellow
0	1	0	2	Blue
0	1	1	3	Red
1	0	0	4	Buff (White)
1	0	1	5	Cyan
1	1	0	6	Magenta
1	1	1	7	Orange

If you look at the column labelled n then you will probably notice that n is one less than the the colour code used in many of the low resolution graphics commands, it is also the decimal equivalent of the three bits C2-C0 if they are read as a single number. Any of the squares that are 'off' show up as black on the display.

What this means is that within any character location any of the four squares can be set to 'on' or 'off'. All the 'on' squares are displayed in the colour given by C0 to C2 and all of the 'off' squares show as black. This obviously places some limitations on the way that low resolution graphics can be used.

Using low resolution graphics

From a consideration of the data format used for low resolution graphics, or simply from experimentation, it is not difficult to discover that any particular character location cannot show more than one colour and black. The low resolution graphics commands SET and RESET, however, do not make this restriction at all clear. For example, the command

SET(x,y,c)

where x,y is the co-ordinate of the point and c is the colour you would like it to be set to, leads you to believe that you can SET any point on the low resolution graphics screen to any of the eight possible colours. What it doesn't tell you is that:

if the point is currently within a text character then SETting it will produce three 'off' or black points around it

if the point is currently part of a low resolution graphics block then all of the other 'on' points will change to its colour.

This means that that there are only two ways to use low resolution graphics without odd effects -

1) SET all the squares with in a character block all 'on' and to the same colour or all 'off'. This effectively reduces the resolution of the screen to 32 by 16 but does give a true nine colour display.

2) Work with a black background (i.e. use CLS 0) and accept the limitation that you can only have one colour within each character block.

The first approach is not really that useful apart from very low resolution games, histograms and kaleidoscope-type patterns. If you intend to use the low resolution screen in this way then then it is much better to ignore the SET and RESET command in favour of using PRINT @ with the graphics character codes described below.

The second approach does give the sort of resolution that is useful but the problem lies in writing your program so that it doesn't try to set more than one colour per character block. Once again, if at all possible, it is better to work with PRINT @ and graphics character codes. An alternative procedure is to use the POINT command to find out what colour is already present at the location but this will mean checking not only the point about to be plotted but the three other points that make up the character block. A better solution is to define a function that will check the screen character location to find out what colour is set and if any points are on.

```
DEF FNC(M)=INT(PEEK(M)/16)-7
DEF FNO(M)=PEEK(M) AND 15
```

The first function FN C will return the colour code of the character location stored in memory location M. The way that it works is to use PEEK to retrieve the contents of memory location M, then 'get rid of' the first four bits of the number by dividing by 16 and taking the INTeger part. Finally it removes bit 7 by subtracting 8 and converts the code to the standard Dragon colour codes by adding 1. If you don't follow this explanation then write down an eight bit binary number and try it! The second function will return a number that is

related to how many of the low resolution graphics points are on. More importantly, it returns 0 if no points are on. It works by PEEKing the memory location and then using the AND function to remove all the bits other than the first four. The use of the AND function in this way will be explained in detail in Chapter 9. Both of these functions will only work if the character location isn't in text mode but text mode can be recognised by the first function returning a negative number. You don't have to understand how these functions work to make use of them and as an example consider the low resolution kaleidoscope program given below -

```
  10 CLS 0
  20 DEF FNC(M) =INT(PEEK(M)/16)-7
  30 DEF FNO(M)= PEEK(M) AND 15
  40 X= RND(31)
  50 Y=RND(15)
  60 C=RND(8)
  70 M=&H400+INT(X/2)+32*INT(Y/2)
  80 IF FNC(M)=C THEN GOSUB 1000
  90 IF FNO(M)=0 THEN GOSUB 1000
 100 GOTO 40
1000 SET(X,Y,C)
1010 SET(63-X,Y,C)
1020 SET(X,31-Y,C)
1030 SET(63-X,31-X,C)
1040 RETURN
```

In lines 10 to 30 the screen is cleared to be completely black and the two functions are defined. Then, lines 40 to 60, an X and Y co-ordinate value that lies in the top righthand quarter of the low resolution screen is generated along with a random colour code. Line 70 works out the memory location that corresponds to X,Y remembering to divide each co-ordinate by two so as to convert from low resolution co-ordinates to text screen co-ordinates. Then the FNC function is called to see if the character location is the same colour as the one about to be SET. If it is, then subroutine 1000 is called to SET the point and three mirror images of it in the other three quarters of the screen. Line 90 checks to see if any of the points are on yet. If not, then subroutine 1000 is called to SET the point. This means that a point will only be SET if it is in a text square with points of the same colour or if it is in a text square with no points SET as yet. The display produced by this program is quite pretty and is a distinct improvement over just SETting random points. To see the difference remove line 80 and replace line 90 with GOSUB 1000 which will then SET the point without any tests.

In general using SET and RESET to create displays with more than one colour and black is difficult. There is a lot to be said for the approach described in the next section where the graphics codes are treated as extra text characters.

Using graphics characters

So far the low resolution graphics facilities of the Dragon have been described in terms of turning points on and off within a character location. This attitude towards low resolution graphics leads naturally to SET and RESET and their associated problems. Now, while there is no avoiding the limitations of the Dragon's low resolution colour graphics, there is another way of thinking about them that, in my opinion, makes things a little easier.

If you think about a character location divided into four smaller squares then you can make a list of all the possible ways of turning these four squares 'on' and 'off'. If you do this you will find that you have a list of 16 different shapes corresponding to the graphics characters listed in Appendix B of the Dragon manual. You can indeed think of these 16 shapes as graphics characters because the CHR$ function will accept numbers corresponding to the code that has to be stored in screen memory to produce each shape. This means that, apart from not being available on the keyboard, they can be treated like the other text characters. There is only one complication - you have to define the colour that each of the 'on' points will take. This leads to one set of 16 graphics characters for each of the eight possible colours. Rather than use a huge table to show which character codes corresponded to which graphics character with which colour, it is simpler to give an equation. The character codes for the 16 graphics characters in green on black are 128 to 143. To see these characters try -

```
10 FOR I=128 TO 143
20 PRINT CHR$(I);" ";
30 NEXT I
```

To obtain any of the 16 shapes in any other color use -

```
CHR$(code+(c-1)*16)
```

where code is the character code of the shape in green and c is the standard colour code of the colour that you want to use.

The advantage of this use of graphics characters is that there is never any attempt to use more than one colour in each character location and the character codes can be stored in BASIC strings so as to make up small shapes. For example, try

```
10 CLS 0
20 S$=CHR$(131)+CHR$(135)+CHR$(139)+CHR$(131)
30 R$=CHR$(128)+CHR$(143)+CHR$(143)
40 T$=CHR$(134)+CHR$(128)+CHR$(128)+CHR$(137)
50 PRINT @160,S$;
60 PRINT @192,R$;
70 PRINT @224,T$;
80 GOTO 80
```

which prints a green man! You can change the colour of the man by altering the character codes using the formula given above and PRINT him anywhere on the screen using the PRINT @ command.

Using the other semi-graphics modes

The Dragon's low resolution graphics are in fact just one of a number of semi-graphics modes. These modes are not available from BASIC but it is quite easy to write a number of subroutines to set the video generator and the SAM chip to produce them. The semi-graphics modes and the corresponding settings of the video generator and SAM chip can be seen in table 3.2.

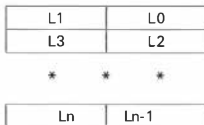
Table 3.2

The Semi-graphics Modes

SAM v2	v1	v0	PIA 1 76543210	resolution rows x cols	colours	memory	graphics mode
0	0	0	0XX0XUUU	32 x 64	9	512	4
0	0	0	0XX1CUUU	48 x 64	2	512	6
0	1	0	0XX0XUUU	64 x 64	9	2048	8
1	0	0	0XX0XUUU	96 x 64	9	3072	12
1	1	0	0XX0XUUU	192 x 64	9	6144	24

Where X means don't care, U means do not change and C selects between one of two colour sets.

Each of the semi-graphics modes works in the same way as the low resolution graphics mode but differs in the number of parts it divides a character location into and the exact details of the memory map. For example, semi-graphics mode 4 divides a character location into four parts and uses one memory location per character location. However semi-graphics 12 dividesa characterlocationinto12 partsan uses sixmemory locations per character location. In general semi-graphics n divides a character location into n parts as shown -



To use one of the semi-graphics modes we need three subroutines -
1) A subroutine to set V0 to V2 of the SAM chip
2) A subroutine to set the PIA bits
3) A subroutine to set and reset any point specified by x and y co-ordinates.
Once we have these, any semi-graphics mode becomes as easy to use as the low resolution graphics mode (which is in fact semi-graphics 4).

The subroutine to set the SAM chip is easy enough -

```
1000 POKE V0 + &HFFC0,0
1010 POKE V1 + &HFFC2,0
1020 POKE V2 + &HFFC4,0
1030 RETURN
```

To use it all you have to do is storethe values forV0 to V2 shown in table3.2 in the variable of the same name. The way that it works should be obvious from the discussion of the SAM chip at the beginning of this chapter but, if you are in doubt, work out what line 1000 does for V0 = 0 and V0 = 1.

The subroutine for setting PIA1 is more difficult because of the need to leave some of its bits unaltered. PIA1's data register is at FF22 hex and it behaves just like any standard Dragon memory location, except of course for the fact that it controls the voltages on eight output lines. (The PIAs are

explained in more detail in Chapter 7.) A subroutine to set the output lines in table 3.2 is given below -

```
2000 A= PEEK (&HFF22) AND 7
2010 A=A+8*CSS+16*G0+32*G1+64*G2+128*AG
2020 POKE &HFF22,A
2030 RETURN
```

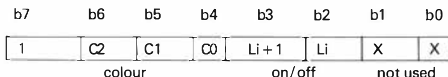
The subroutine is used by setting the values of CSS,G0,G1,G2 to the values given in the table. The way that it works is to first read the current setting of the PIA data register and 'mask' off the bottom three bits so that they can be preserved (see Chapter 7) Line 2020 then constructs the value given by the new states of the lines that we are changing and the old states of the lines that we are leaving unchanged. This if finally POKED back into the data register.

Using these two subroutines it is simple to enter any of the semi-graphics modes and all that is we need is a subroutine that will set and reset points. There are two ways of doing this. We could either write a subroutine for each mode or we could try and write one subroutine that would take the mode into account. If semi-graphics mode 4 and mode 6 are ignored then the other modes fall into a very regular pattern. As semi-graphics mode 4 is normal low resolution graphics and semi-graphics mode 6 is only two colour there is little to be lost if the set/reset subroutine only works with the other modes. A general set/reset subroutine for semi-graphics mode 8 to 24 is given below -

```
3000 M=&H400+INT(X/2)+32*Y
3010 X1=1-X+INT(X/2)*2
3020 Y1=Y-INT(Y*2/MODE)*MODE/2
3030 S=1
3040 C1=C-1
3050 IF C=0 THEN S=-1:C1=0
3060 N=128+16*C1+(PEEK(M) AND 15)
3070 F=4
3080 IFY1>MODE/4 THEN F=1
3090 IF S=1 THEN N=N OR (F*(X1+1)) ELSE N=N AND
     NOT(F*(X1+1))
3100 POKE M,N
3120 RETURN
```

To use this subroutine all you have to do is set MODE to the correct mode, X to the x co-ordinate Y to the y co-ordinate and C to the colour that you want the point to become. If C is 0 then the point is reset and if C is greater than zero

the point is set. All you have to remember is to use the earlier pair of subroutines to place the Dragon in the correct semi-graphics mode before using it. The subroutine is a little too complicated to explain in detail but if you study it you should be able to make sense of it. There are two components to the subroutine, finding out where to store the screen data and working out what should be stored there. Line 3000 works out the position in memory that corresponds to the screen position X,Y. The rest of the subroutine is mostly about working out the value that should be stored there. Each memory location sets the colour and defines the state of just two points using the following format for positions within the top half of a character location -



In the lower half of the character location the format is roughly the same except that the on/off information is stored in bits0 and 1 and bits2 and3 are unused. Li+1 and Li refer to two points on the same horizontal 'line' in the character location as shown earlier. As there is now more than one memory location corresponding to each character location the memory map is a little more difficult. Instead of storing all of the information concerning one character and then the next and so on, the memory map is such that the top row of points from the top row of characters is stored first, then the second row and so on until all the information about the top row of characters has been stored. This means that, if you forget about character locations for a moment, the memory location corresponding to a point x,y on the screen is simply &H400 + INT(x/2) + 32*y. Although both the memory map and the data formats are quite complicated you should be able to follow the logic of subroutine 3000 after a few attempts.

Even if you don't make the effort to understand the three subroutines given above, you can still use them. As an example consider the problem of implementing the kaleidoscope program given earlier but this time using semi-graphics24 for increased resolution. The first thing that we have to do is set the Dragon into semi-graphics24 using subroutines 1000 and 2000. If you look at table 3.2 you can see that to obtain this mode V0=0,V1=1 and V2=1, also all of the PIA bits are either 0 or X (i.e. don't care) so we might as well set all of the PIA graphics bits to zero. So the kaliedoscope program starts -

```
10 V2=1:V1 =1:V0=0
20 GOSUB 1000
30 CSS=0:G2=0:G1=0:G0=0:AG=0
40 GOSUB 2000
```

If you type in this program along with subroutines 1000 and 2000 and an extra line, 50 GOTO 50, to stop the Dragon reverting to text mode you will see, from the garbage displayed, that we also need a screen clear routine. You could clear the screen by using subroutine 3000 to write a coloured block to each screen location but this is slow. The following subroutine will POKE a semi-graphics 'all points off' code into every memory location in the screen area -

```
4000 FOR I=&H400 TO &H400+6144
4010 POKE I,128
4020 NEXT I
4030 RETURN
```

We can now return to the main program. After setting the mode and clearing the screen we can get on with drawing the kaliedoscope. This is nothing more than a direct translation of the earlier program but using subroutine 3000 instead of the SET command and making allowance for the higher resolution.

```
50 GOSUB 4000
60 MODE=24
70 X2=RND(31)
80 Y2=RND(95)
90 C=RND(7)+1
100 X=X2:Y=Y2:GOSUB 3000
110 X=63-X2:Y=Y2:GOSUB 3000
120 X=X2:Y=191-Y2:GOSUB 3000
130 X=63-X2:Y=191-Y2:GOSUB 3000
140 GOTO 70
```

Notice the way that X and Y are set to the co-ordinate of the point that is to be set to colour C before the plotting subroutine is called. If you have typed in all the parts of this program (main program lines 10 to 140 and subroutines 1000, 2000, 3000 and 4000) then you will be rewarded with a high resolution kaleidoscope in nine colours! It has to be admitted that clearing the screen and plotting so many points takes rather a long time in BASIC but it is not too bad for many applications. To make sure that you understand how to use the other semi-graphics modes try altering this program to work in semi-graphics mode 8 and 12. All you have to do is alter lines 10,30 and 60 and change the limits of the Y co-ordinate to reflect the lower resolution.

When to use Semi-graphics

The Dragon's semi-graphics modes are indeed a bonus! The standard low resolution graphics mode using semi-graphics mode 4 is good for many purposes and is the easiest way to have nine colours on the screen. The main attraction of the unsupported semi-graphics modes is the increased resolution and the continuing availability of the full nine colours. The Dragon's full graphics modes provide higher resolution than semi-graphics 24 but none of them provide more than four colours. The only disadvantage of the semi-graphics mode is the restriction on the way that the colours can be used and this was extensively discussed with reference to BASIC's low resolution graphics. The ways around this problem generalise to the other semi-graphics modes and it is safe to say that the easiest way to use semi-graphics 24 is with a black background! Although the subroutines given in this chapter do allow you to use these hidden modes from BASIC they are even more useful to the machine code programmer. If you know machine code you should have little trouble in translating the subroutines into 6809 assembly language and then the semi-graphics modes are all yours, without the speed restrictions of BASIC.

Chapter Four

High Resolution Graphics

In the last chapter the Dragon's text and low resolution graphics screen was described along with a number of new, normally hidden, semi-graphics modes. This theme of discovery continues in this chapter where the Dragon's true or full graphics modes are examined and a few new ones are explained. Once again if you are uninterested in the technical details you can simply use the results by reading the technical summaries and practical sections. To get the most from this chapter it is important that you have read Chapter 3.

Free graphics but no alpha!

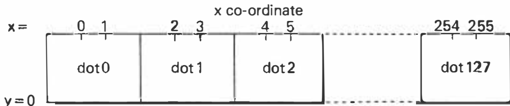
Of the three ways of working that the graphics chip offers us, the true graphics modes are the most flexible. Unlike text mode where only alphanumeric characters can be displayed, and semi-graphics modes where points can be plotted but with restrictions on colour, the full graphics modes provide complete control over what appears on the screen. In graphics mode you can change any point to any of the available colours without reference or change to any other point. This freedom comes at a price, however, and in general graphics modes provide fewer colours for more memory. Indeed the largest number of colours that can be used in a graphics mode is four. Another disadvantage is that there is no automatic way of producing alphanumeric characters on a graphics screen. However, even with these two problems the Dragon's graphics modes are often more useful than any others.

Using high resolution graphics

There are five high resolution graphics modes supported by Dragon BASIC -

PMODE	resolution row x col	number of colours	memory used
0	96 x 128	2	1536
1	96 x 128	4	3072
2	192 x 128	2	3072
3	192 x 128	4	6144
4	192 x 256	2	6144

These are, in fact, just five out of eight possible graphics modes offered by the video generator and SAM in combination but more of this later. If you look at the resolutions offered by the high resolution modes you will see that the horizontal resolution is either 128 or 256 and the vertical resolution is either 96 or 192. However, no matter what PMODE you are working in the Dragon's high resolution graphics commands always assume that the screen is 256 wide by 192 high. This is an advantage because it means that you can write a program in one PMODE and if you don't like the result you can try it in another PMODE without having to change all the co-ordinates to take account of the change in resolution. On the negative side, working with a co-ordinate system that leads you to believe that you have more points than you actually have, can produce some odd effects. If you are working in PMODE 4 then there is no problem. Changing the x or y co-ordinate by 1 moves you on to a new dot on the screen so you can quite happily write programs that change a dot at x,y to one colour and a dot at x+1,y or x,y+1 or even x+1,y+1 to a different colour and you will see both dots. But in the lower resolution modes the dot specified by x,y and x+1,y could be the same dot! For example in PMODE 2 or 3 there are only 128 dots horizontally. If you start at the top left hand corner then 0,0 and 1,0 both specify the SAME dot but 1,0 and 2,0 refer to different dots -



This is simply a result of there being twice as many different x co-ordinates as there are horizontal dots! In PMODEs 0 and 1 the same thing happens with the y co-ordinate as there are twice as many y co-ordinates as vertical dots.

Much of the time, this excess of co-ordinates causes no real problems. If you are plotting a shape and you specify it more accurately than the PMODE can display it, you are only wasting a little time. However, sometimes odd things can happen. Try the following short program -

```
 10 PMODE4
 20 SCREEN 1,0
 30 PCLS 0
 40 C=0
 50 FOR Y=0 TO 191
 60 FOR X=0 TO 255
 70 PSET(X,Y,C)
 80 IF C=0 THEN C=1 ELSE C=0
 90 NEXT X
100 IF C=0 THEN C=1 ELSE C=0
110 NEXT Y
120 GOTO 120
```

What it does is quite straightforward. After setting up PMODE4 in lines 10 to 30 it PSETs every point on the screen to either black or green. Lines 80 and 90 look a little odd but all they do is 'flip' the value in C - i.e. if C is 0 it is changed to 1 and if it is 1 it is changed to 0! This results in adjacent points taking up opposite colours. The pattern of colours on the first line is black,green,black,green and so on. Because of line 100 the pattern on the next line starts with green, the next with black and so on. The result is an extremely fine chequered pattern in black and green. This program works only because there are as many points in PMODE 4 as there are co-ordinates. Now try changing line 10 to PMODE2 and re-running the program. You will now find that rather than a chequered pattern you get horizontal stripes. If you change line 10 to PMODE 0 then you will find that although things do change on the screen the final result is the solid black display that you started with. I leave it to you to explain these two results but the cause is obvious if you watch the way the colours change on the screen and think about the fact that sometimes different co-ordinates specify the same dot. Other examples of this effect will be given in Chapter 6 in connection with the DRAW command.

Apart from offering different resolutions, the different PMODEs also offer different colour ranges. In fact each PMODE can use one of two colour sets. The two colour modes can select between -

Set 0 Colour	code	Set 1 Colour	code
Black	0	Black	0
Green	1	Buff	5

and in the four colour modes -

Set 0 Colour	code	Set 1 Colour	code
Green	1	Buff	5
Yellow	2	Cyan	6
Blue	3	Magenta	7
Red	4	Orange	8

To display any graphics in the colour set of your choice all you have to do is SCREEN 1,0 for set0 and SCREEN 1,1 for set 1. It is important to realise that you can change the colours of existing graphics from one colour set to the other without having to redraw anything. For example try -

```
10 PMODE3
20 COLOR 4,3
30 PCLS
40 LINE (50,50)-(100,100),PSET,BF
50 SCREEN 1,0
60 FOR I=1 TO 1000:NEXT I
70 SCREEN 1,1
80 FOR I=1 TO 1000:NEXT I
90 GOTO 50
```

All that this program does is to draw a square on the screen and then select colour set 0 and then colour set 1 and so on. Notice that red changes to orange and blue to magenta. The correspondence between colours in the two sets is obvious.

In a two colour mode the most useful colour set is probably set 1 giving a black on white (buff) or a white on black display. However, in general set 0, green and black, produces a sharper picture than set 1. In a four colour mode the choice is between a colour set that includes bold colours, i.e. set 0 with

red, blue and yellow and one that includes white. If you need white then there is no choice but to use set 1 along with its weaker colours.

Finally, before moving on to consider the details of how high resolution graphics works and the missing modes, it is worth going over the exact meaning and use of the high resolution initialisation commands, PCLEAR, PMODE, SCREEN, COLOR and PCLS.

PCLEAR is used to reserve memory for use by high resolution graphics. It is not normally needed unless you intend to used paged graphics (see Chapter 6) or really need to reclaim some of the memory that BASIC normally reserves for the high resolution display. In any case it should only be used as the very first command in a BASIC program.

PMODE serves a dual purpose. It sets the high resolution graphics mode that all subsequent high resolution commands work in and it can be used to set the 'start page' for the display. Once again setting the 'start page' is something that is not often used except for paged graphics (see Chapter 6). The most usual form of the PMODE command is PMODE n where n is the number of the mode that you want to use. Notice that PMODE doesn't switch the Dragon into the requested display mode, it simply causes all of the subsequent graphics commands to write into the area of memory used by the mode.

SCREEN determines what is actually displayed and which colour set will be used. SCREEN 1,s will cause the graphics screen mentioned in the last PMODE command to be displayed using colour set s. SCREEN 0,s will cause the text screen to be displayed. Specifying s=0 gives the usual black on green and s=1 gives black on orange. The most important point to notice is that SCREEN is the only command the changes the Dragon from one display mode to another.

COLOR f,b sets the colours that will be used as foreground and background colours. It is important to note that COLOR doesn't actually change any colours already on the screen it simply sets the colours to be used by subsequent commands. In particular changing the background colour will not suddenly change the colour that everything is displayed against!

PCLS c clears the screen by filling it with colour c. If you do not specify c then the current background colour will be used.

The standard way of starting a high resolution colour program is
1) set the PMODE that you want to work with e.g. PMODE 4
2) do anything that you want to do before the high resolution screen is displayed e.g. PCLS 2
3) set default background and foreground colours, e.g. COLOR 3,2
4) display the screen in the colour set of your choice, e.g. SCREEN 1,0

Notice that there is no need to use a PCLEAR or specify a start page in the PMODE command unless you are short of space or involved in paged graphics. Step 2 can be much longer than the simple example. You could choose to complete a complicated piece of graphics out of the user's view and then suddenly show it by using a SCREEN command.

The Dragon's graphics modes

The technical details of the Dragon's full graphics modes are very similar to the details of semi-graphics described in the last chapter. The settings of the video generator and the SAM chip can be seen in table 4.1 along with details of the modes they produce -

Table 4.1
The full graphics modes

SAM			**PIA 1**	**resolution**	**colours**	**memory**	**graphics**
v2	**v1**	**v0**	**7 6 5 4 3 2 1 0**	**rows x cols**			**mode**
0	0	1	1000CUUU	64 x 64	4	1024	1F
0	0	1	1001CUUU	64 x 128	2	1024	IT
0	1	0	1010CUUU	64 x 128	4	2048	2F
0	1	1	1011CUUU	96 x 128	2	1536	PMODE 0
1	0	0	1100CUUU	96 x 128	4	3072	PMODE 1
1	0	1	1101CUUU	192 x 128	2	3072	PMODE 2
1	1	0	1110CUUU	192 x 128	4	6144	PMODE 3
1	1	0	1111CUUU	192 x 128	2	6144	PMODE 4

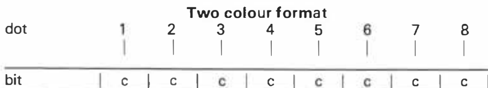
where U means do not change and C selects between one of two colour sets. If you look at this table you will see that unlike the semi-graphics modes Dragon BASIC only ignores three of the true graphics modes. It is still worth finding out how to use these extra modes and on the way to discover the format and memory map used by the more familiar PMODEs.

Using the full graphic modes

To make use of the modes that BASIC ignores or to make direct use of the PMODEs we need three subroutines. The first two are exactly the same as the ones used to set the SAM chip and the video generator given in the previous chapter i.e. subroutines 1000 and 2000. The third subroutine, to plot any point in any colour, is the only one that has to be re-written taking into account the new memory maps and data formats that the graphics modes use.

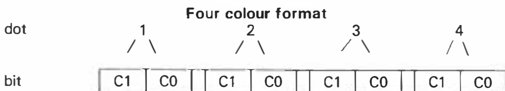
The graphics data formats are straightforward enough. In a two colour mode each memory location controls the colour of eight points in a row on the screen. The colour of each point is simply selected by whether each bit is a 1 or a 0. For example, a memory location containing 15 decimal, that is 00001111 in binary, would produce a row of eight dots, four in each colour. The data format for a four colour mode is almost as simple in that each memory location now controls the colour of four points on the screen. The only complication is that the colour of each point is determined by two bits. (To select one of two colours only needs one bit but to select one of four colours needs two bits!) The bits are grouped in pairs in the most obvious way so if a memory location contained 00011011 the grouping would be - |00|01|10|11| defining four dots in colour 0, 1, 2, and 3 respectively. These two formats can be summarised as -

Two colour format



where each C = 0 or 1 and determines the colour of the dot

Four colour format



where each pair of bits C1,C0 determine the colour of one of the dots.

The memory map for all the modes is equally simple. The address of the memory location that determines the colour of the point at x,y is -

m = start address + INT(x*n/16) + xres*n/16*y

where *start address* is the address of the first screen location (this is normally 0600 hex but more of this later), n is the number of colours available in the mode, and xres is the x resolution (i.e. the number of columns) in the mode. Using all this information we can now write a BASIC subroutine to set any point to any colour in any graphics mode.

```
3000 P= 16/N
3010 M=SA+INT(X/P)+XRES/P*Y
3020 X1=P-1-X+INT(X/P)*P
3030 X1=X1*8/P
3040 C1=2↑(8/P)-1
3050 D=PEEK(M) AND NOT(C1*2↑X1)
3060 D=D OR (C*2↑X1)
3070 POKE M,D
3080 RETURN
```

This subroutine can be use by setting N to the number of colours in the mode, ES to the number of columns, SN to the start address of the display, X to the x co-ordinate, Y to the y co-ordinate and C to the colour (0-3) that you want the point to be.

Before we can make use of any of this we need one more subroutine. In the previous chapter we allowed the start of the semi-graphics displays to be the same as the start of the text screen. In fact we can make the video generator display almost any area of the Dragon's memory. To do this we have to use some new locations within the SAM chip. The group of 14 addresses from FFC6 to FFD3 are arranged in pairs and work together in roughly the same way that the pairs that we called V0 to V2 do. That is, writing to an odd address clears the condition and writing to an even address sets it. The seven pairs of addresses are usually called S0 to S6 and can be thought of as a seven bit binary number with 'clear' as 0 and 'set' as 1.

Name	address	bit
S6	FFD3/FFD2	6
S5	FFD1/FFD0	5
S4	FFCF/FFCE	4
S3	FFCD/FFCC	3
S2	FFCB/FFCA	2
S1	FFC9/FFC8	1
S0	FFC7/FFC6	0

To make the video generator display the memory starting at S all you have to do is to set or clear S0 to S6 according to whether or not the corresponding bits in the binary representation of S/&H200 are one or zero. The reason that you have to divide by &H200 is simply that the screen memory can only start at multiples of 1/2K (&H200) memory locations. Perhaps the simplest thing is always to use the following subroutine:

```
5000 SN=INT(S/&H200)
5010 FOR I=0 TO 5
5020 POKE &HFFC6+I*2+(SN AND 1),0
5030 NEXT I
5040 RETURN
```

This subroutine will POKE the SAM chip so that the display memory starts at the 1/2K boundary below the address contained in S. You can use this subroutine to set the display area for ANY mode.

As an example of using these subroutines consider the problem of plotting random points in the full graphics 64 x 64 mode. The program begins by setting the SAM and the video generator to the correct mode.

```
10 V2=0:V1=0:V0=1
20 GOSUB 1000
30 AG=1:G2=0:G1=0:G0=1:CSS=0
40 GOSUB 2000
```

The next step is to set the starting address of the area of memory to be displayed - &H600 is the usual start for high resolution graphics -

```
50 S=H&600
60 GOSUB 5000
```

Now we can plot random points using subroutine 3000, remembering to set N and XRES first -

```
 70 N=4
 80 XRES=64
 90 C=RND(3)
100 X=RND(63)
110 Y=RND(63)
120 GOSUB 3000
130 GOTO90
```

If you run this entire program (remembering to include subroutines 1000 and 2000 from the previous chapter and subroutines 3000 and 5000 given above)

you will see a 64 x 64 four colour display with out any of the limitations that you find in semi-graphics modes.

As an exercise you might like to try to turn these random points into a four colour kaleidoscope and then see what the program looks like in the other graphics modes.

The best of all possible modes

The Dragon has so many display modes that it's not really surprising the BASIC ignores some of them. Each of the modes has advantages and disadvantages and it can be difficult to know which one to use for any given application. If you are programming in BASIC then my advice is to experiment with the new modes that have been introduced in this chapter but do not plan to use them for any sizeable program until you are entirely familiar with their behaviour. The Dragon's extended graphics commands are not something to be given up lightly. If you are programming in assembler then there is no such luxury to give up and you are free to choose whatever mode you like! My own preferences are to use the 64 by 64 full graphics mode for low resolution graphics and select the most appropriate PMODE for high resolution graphics. If you really feel the need for nine colours then select the semi-graphics mode that gives the required resolution but remember that this makes any program much more complicated and, in general, semi-graphics are best avoided. There is no denying that the full graphics modes are the easiest to work with and they would be the best if only there was a simple way to display text and if a better choice of colours was available. Unfortunately there is nothing that can be done about the choice of colours without a soldering iron but there is a way around the 'no text' problem, as Chapter 6 will show.

CHAPTER FIVE

Dragon Sound

The Dragon's sound generation is both simple and sophisticated. Unlike many other microcomputers it doesn't use a special sound effects chip but instead uses six of the output lines of one of the PIAs to create waveforms directly. In technical terms, the PIA is used as a 6 bit Digital to Analog convertor (D to A). This method has the advantage of being cheap and flexible in that the wave forms that come out of the D to A can be changed using nothing but software. However, although Dragon BASIC provide two very powerful sound commands SOUND and PLAY there is still room for improvement. The D to A convertor has other uses apart from sound generation. It produces the coded signal that is recorded when ever you CSAVE a program and it also plays a major role in reading the joystick inputs. If you are technically oriented then you could use the output from the D to A as a wave form generator but you would need to bear in mind the low quality of the output.

The Dragon's D to A convertor is very useful but it is only one of four possible sound sources. Of the others, the first is the cassette tape recorder, used to save and load programs, which can also be selected in such a way that its audio output is reproduced over the TV's loudspeaker. The second is an external sound source, normally coming from a program cartridge, although almost anything can be played over the TV set using this input. Finally, there is a 'single bit' sound source that can be used to supplement the range of sounds that the Dragon can make. This single bit sound source also has one other use in that it can work as a sound detector! You might not be able to see a reason for needing to know when a Dragon sound starts or stops, after all surely if the Dragon produces the sound it 'knows' when it starts and stops. By the end of this chapter you will understand just how useful this can be.

To make use of any of the Dragon's more specialised sound facilities there

is no avoiding a certain amount of technical discussion about the internal workings. However, the chapter begins by taking a fresh look at the BASIC sound commands.

Using SOUND and PLAY

The simplest sound-producing command is -

SOUND p,d

where p is a number between 1 and 255 specifying the pitch of the tone and d is a number greater than 0 specifying the duration of the tone. This is such a simple command that it is difficult to imagine that there is anything to add to its description. However, the Dragon manual doesn't tell you what values of p and d to use to produce any given note for any given time. All it does tell you is that middle C corresponds to a value of p of 89. To a certain extent this is forgivable in that the PLAY command is available for producing musical notes. However, it is sometimes useful to be able to use SOUND to generate notes of a given duration or of a given pitch.

The d parameter actually specifies the duration of the sound in 4/50ths of a second. So a d value of 25 will make a sound for 2 seconds and a value of 12 will make a sound for around 1 second. The p parameter controls the pitch in a fairly straightforward manner but unfortunately when this is coupled with the details of the musical scale things get a little complicated. If you want to produce a sound with a frequency of F then the value of P that you would have to use is given by -

$$P = 256 - (167/r) + 5*(1-1/r)$$

where r is the ratio of the frequency with the frequency of middle C In other words if Fc is the frequency of middle C then

$$r = F/Fc$$

(Middle C is defined as 261.6Hz on the tempered scale but to use the above equation for accurate frequency generation you would have to check what your Dragon actually produced for SOUND 89,d.) As the ratio of notes a semi-tone apart is 2↑(1/12) the above equation can be used to specify notes in terms of the number of semi-tones that they are above or below middle C. Try the following -

```
10 DEF FNN(P)=256-167/2↑(P/12)-5*(1/2↑(P/12)-1)
20 FOR I=0 TO 12
30 SOUND FNN(I),8
40 NEXT I
```

The function FNN(P) will return the pitch number required to produce any note if P is the number of semi-tones that the note is above or below middle C. This function can be used to produce music in the range corresponding to an octave above what you can manage using the PLAY command. As another example try -

```
10 DEF FNN(P)=256-167/2↑(P/12)-5*(1/2↑(P/12)-1)
20 S=S+SGN(RND(0)-.5)
30 IF S<.7 THEN S=.7
40 SOUND FNN(S),1
50 GOTO 20
```

this plays notes that randomly change by plus or minus one semi-tones and sounds very interesting!

The PLAY command is described at great length in the Dragon manual and when it comes to transcribing tunes from sheet music to the Dragon it is difficult to think of anything more convenient. However it is worth pointing out that the PLAY command can be used to create sound effects as well as music. The principle is simply to play a string of notes at a very high tempo. For example try -

```
10 PLAY "T155V31O1CGCGV20CGCGV10CGCGCGV5CGCGCG"
20 IF INKEY$ =" "THEN GOTO 20
30 GOTO 10
```

for the sound of a gun firing. The PLAY command in line 10 may look complicated but it is in fact made up of the same section repeated. The first part of the PLAY string, "T155V31", sets the tempo and the volume to maximum.Thenextpart,"O1CGCG",setstheoctavetothelowestpossibleand plays the note pair CG twice. Then this note pair is then repeated at decreasing volume until the end of the string. The whole effect works because of the speed with which the notes are sounded. If you want to hear what the PLAY string sounds like played normally change T155 to T5 and the effect will vanish! You can produce a wide range of sound effects using the PLAY command in this way. The only trouble is that it takes quite some time to change the play string by trial and error to make the sound that you desire.

A problem common to both SOUND and PLAY is that while any sound is being produced BASIC stops running. This isn't too much of a problem as long as you are not trying to animate something on the screen while making appropriate sound effects. In this case the best that you can do from BASIC is

to move the object and then make some sound, then move the object again and so on. It isn't really possible to do any better if you choose to use a language other than BASIC because the Dragon can only do one thing at a time - either move a shape on the screen or make a noise. However if you use a faster language such as assembler then the swapping between the two actions, moving and making a noise happens so quickly that the Dragon appears to be doing both at the same time. This problem of not being able to make a noise while doing something else is a result of the Dragon not using a special sound effects chip that can get on with sound generation all on its own.

The sound generator - a 6 bit D to A.

The Dragon's sound generator is in fact nothing more than 6 output lines of one of the PIAs wired together in such away that you can produce a range of output voltages. A single output from a PIA is capable of assuming one of two possible states - high corresponding to approximately 5 volts and low corresponding to approximately 0 volts. Which of these two states it adopts is controlled by setting a bit in the PIAs data register either to 1 for high or to 0 for low. You can see that this allows us to control the voltage on an output line using nothing but software to set and clear bits in the data register. However, this wouldn't be very much use because we are restricted to one of two voltages whereas to produce a pure sounding tone we need to be able to produce a range of voltages.

The solution adopted by the Dragon is very clever indeed. The output from each of six output lines of PIA1 are added together to produce a final output voltage. If we number the output lines 0 to 5 then instead of each being able to contribute it full range of 0 or 5 volts the outputs are reduced so that each one can only produce half the voltage of its higher numbered neighbour. For example if line 5 can produce 0 or 4 volts, then line 4 could produce 0 or 2 volts, line 3 could produce 0 or 1 volt and so on down to line 0 that could only produce 0 or 1/8th of a volt. This may sound like a complicated way of doing things but, if you recall that this is exactly the way binary number works, you will see the reason why. In a six bit binary number each of the bits increases in value by a factor of two as you move from the right to the left. So, for example, the number 011010 is -

	32	*16*	*8*	*4*	*2*	*1*
	*(2*16)*	*(2*8)*	*(2*4)*	*(2*2)*	*(2*1)*	
	0	1	0	4	1	1
26 =	0*32	1*6	1*8	0*4	1*2	1*1

If the PIAs output lines were turned on in the same pattern as the bits in this binary number then when their different outputs were added together the result would be a voltage proportional to 26. That is -

voltage = 0*4 + 1*2 + 1*1 + 0*.5 + 1*.25 + 1*.175 = 3.425

Similarly, setting the output lines to any pattern of bits produces an output voltage proportional to the binary number that the bits represent. This is the principle of a D to A convertor. A binary number (Digital) is converted to a voltage (analog) proportional to the size of the number.

The electronic details of the D to A convertor can be seen in Fig 5.1. The ouputs from the PIA are first buffered by a 4050 CMOS buffer and are then applied to a potential divider composed of six resistors. If you look at the values of these resistors (10K, 20K, 40K, 80K, 160K and 320K) you will be able to see the way that the different outputs are weighted in multiples of two. Obviously the accuracy of the D to A convertor depends on the accuracy of these resistors and in the Dragon they are all 1% tolerance. The exact voltage output is modified by the presence of the 100K and the 68K and 33K resistors which tend to limit the voltage range to less than 0 to 5 volts. The voltage produce by the D to A is approximately given by -

Voltage = (N*0.072) + 0.25 V

where N is the six bit value (0 - 63) used to set the output lines of the PIA. Notice that this equation is only to be taken as a guide to the voltage that the D to A produces because its exact value depends on the accuracy of the resistors and the 5 volt power supply to be found in any particular Dragon.

Now that we know how the D to A works the next step is to try to use it directly. However, before we can do this it is necessary to 'enable' the sound output to the TV set, otherwise our attempts will remain unheard. The selection of the sound output is described in detail in the next section but suffice it to say that two more output pins of a PIA are used to select which

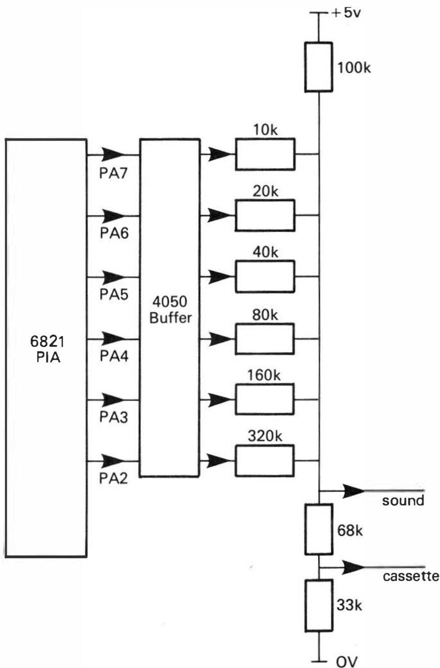


Fig 5.1 The sound generator or 6 bit D to A

sound source is sent to the TV and a third is used to enable and disable the output. The subroutine given below will select and enable the sound output from the D to A convertor. Don't worry about how it works for the moment, all will be clear after the next section.

```
1000 A= PEEK(&HF001) AND NOT(&H8)
1010 POKE &HF001,A
1020 A = PEEK(&HF003) AND NOT(&H8)
1030 POKE &HF003,A
1040 A = PEEK(&HFF23) OR &H8
1050 POKE &HFF23,A
1060 RETURN
```

Once the D to A has been selected and enabled all that is left is to set its output. This is only complicated by the fact that bits 2 to 7 of PIA1 are used and this means that if we want to set the output to V we have to POKE V*4 into the data register to 'avoid' using bits 0 and 1.

```
2000 POKE &HFF20,V*4
2010 RETURN
```

We can now make the output of the D to A take on any value that we like. However because BASIC is so slow it is still difficult to produce any useful sounds. Try -

```
10 GOSUB 1000
20 FOR V=0 TO 63 STEP 4
30 GOSUB 2000
40 NEXT V
50 GOTO20
```

which will produce a periodic 'ramp' waveform or

```
10 GOSUB 1000
20 V=0
30 GOSUB 2000
40 V=63
50 GOSUB 2000
60 GOTO 20
```

which will produce a 'square wave'. You will soon discover that the highest frequency that you can produce is very low! However, it is important that you realise that this is entirely due to BASIC's lack of speed. If you were to use the same techniques in machine code then you could, in theory at least, produce almost any sound by programming the PIA to produce its waveform.

It is also worth knowing that the output from the D to A convertor is always available from the cassette output (pin 5). This means that you can record any music or signals that are played over the TV set by switching the cassette recorder to 'RECORD'. It also means that you could take this output and use it for what ever purpose you wanted! You could for example use your Dragon as a general purpose pulse generator if you added an amplifier and some software. From most user's point of view perhaps the most important thing to realise about the sound generator is that as a general purpose D to A convertor it can be used to create almost any sound - all you need is the software.

Before moving on to consider other aspects of Dragon sound it is worth mentioning that the BASIC sound commands and CSAVE use the D to A convertor to produce good approximations to pure tones. The two tones used to record data by the CSAVE Command are good approximations to a sine wave but as can be seen from Fig 5.2 the approximation used by the sound commands is not so good! However this rough and ready sine wave serves the purpose very well because too pure a tone would soon become boring.

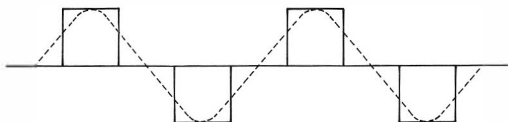


Fig 5.2 Sine Wave Approximation

Selecting the source

The Dragon can select one of three different sound sources - the D to A convertor discussed in the last section the cassette input and an external sound source.

The way that this selection is carried out is quite simple. Two of the 'special' output lines of PIA0, CA2 and CB2 are treated as a two bit number and used to select one of the three sources. Also output CB2 from PIA1 is used as a sound enable/disable bit. The circuit diagram of the selector can be seen in Fig 5.3. Changing the state of these special PIA lines is a little more tricky than for the ordinary output lines as each one corresponds to bit 3 in a different memory location.

PIA0 CA2 is at &HFF01
PIA0 CB2 is at &HFF03
PIA1 CB2 is at &HFF21

A subroutine to set these three bits to a particular state has already been introduced in the previous section. It is just a special case of -

```
1000 A= PEEK(&HF001) AND NOT(&H8)
1010 POKE &HF001,A OR (1 AND S)*8
1020 A = PEEK(&HF003) AND NOT(&H8)
1030 POKE &HF003,A OR (2 AND S)*8
1040 A= PEEK(&HFF23) AND NOT(&H8)
1050 POKE &HFF23,A OR (1 AND E)*8
1060 RETURN
```

which will set the sound source to S and either enable it or disable it depending on the value of E (E = 0 disables and E = 1 enables). The sound source numbers are -

S source
0 D to A
1 Cassette
2 External
3 silence!

To check that this subroutine works as you would expect, place a music tape playing in the cassette recorder and try-

```
10 MOTOR ON
20 S = 1
30 E = 1
40 GOSUB 1000
50 TIMER = 0
60 IF TIMER< 1000 THEN GOTO 60
70 E = 0
80 GOSUB 1000
90 TIMER = 0
```

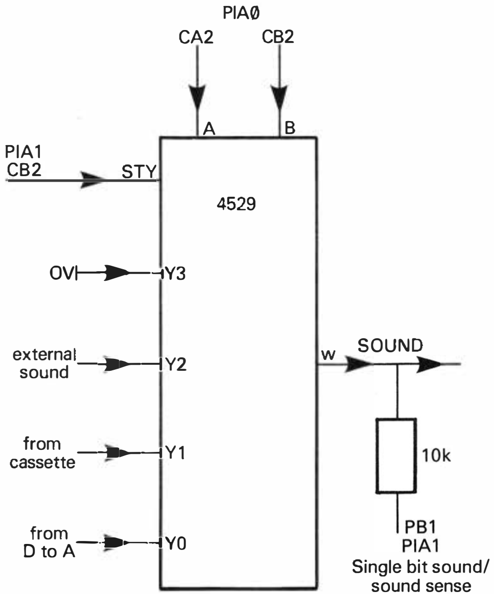


Fig 5.3 Sound source Selection Circuitry

```
1000 IF TIMER< 100 THEN GOTO 100
110 GOTO20
```

This selects the cassette as sound source and then enables it for a while then it disables it and so on.

As the sound commands SOUND and PLAY both select the correct source automatically and BASIC provides the AUDIO command to select the cassette you might suppose that the above subroutine is redundant. However it does serve a useful purpose, as well as being an example, in that it allows you to select the external sound source. This normally comes from a program cartridge plugged into the expansion port but if you connect any suitable sound source to pin 35 of the expansion socket you can select it with subroutine 1000.

Single bit sound and sound sense

There is one other sound source that the Dragon has at its disposal - single bit sound. This is simply a PIA line that is connected directly to the output of the selector described in the last section. It is in fact line PB 1 of PIA1 and it serves two different purposes. Firstly, when set to act as an input it can be used to detect when sound is being sent to the TV set and, secondly, when programmed as an output it can be the promised alternative sound source. Normally it is set to be an input, so to use it as a sound source the first job is to change it to be an output. How to set lines to inputs or outputs is described fully in Chapter 7 so for the moment don't worry about subroutine 1000 given below which changes PB1 from input to output. Once it is set to act as an output there are only two things that we can do - set it to 1 or set it to 0. By changing its state sufficiently rapidly we can produce a very rough sound -

```
10 GOSUB 1000
20 B=0
30 GOSUB 2000
40 B=1
50 GOSUB 2000
60 GOTO 20

1000 A=PEEK(&HFF23) AND NOT(4)
1010 POKE &HFF23,A
1020 A=PEEK(&HFF22) OR 2
1030 POKE &HFF22,A
```

```
1040 A=PEEK(&HFF23) OR 4
1050 POKE &HFF23,A
1060 RETURN
2000 A=PEEK(&HFF22) AND NOT(2)
2010 POKE &HFF22,A OR 2*B
2020 RETURN
```

Subroutine 2000 sets PB1 high and low alternately. Once again the comments about how slow BASIC is are appropriate here. If you want to make any good use of the single bit sound source then you really need to use assembler.

It is in its other role that the single bit sound source is really useful as a sound sensing bit. In its normal state as an input it can be read to discover if there is any sound being sent to the TV set. The reason why you might what to know if sound is being sent to the TV set is revealed in the next section but for now the following user defined function will return 0 if all is silent and 1 if there is any sound.

```
10 DEF FNS(X)=1-(PEEK(&HFF22) AND 2)/2
```

Notice that X is a dummy variable and is only included because the Dragon will give an error message without it.

Using AUDIO and MOTOR

There are two BASIC commands, AUDIO and MOTOR, which when used in conjunction with the sound sensing function can allow your Dragon to do remarkable things. The command MOTOR ON will start your tape recorder running and MOTOR OFF will stop it. (All this assumes that you are using the motor control connection on the tape recorder lead.) The command AUDIO ON will connect the output of the tape recorder to the TV set's sound channel and AUDIO OFF will disconnect it. If you have never tried playing with these commands place a music cassette in your tape recorder, press the 'play' button and then try -

```
10 PRINT "PRESS 'H' TO HEAR"
20 PRINT "PRESS 'S' TO STOP;;
30 MOTOR ON
40 A$=INKEY$
50 IF A$="H" THEN AUDIO ON
60 IF A$="S" THEN AUDIO OFF
70 GOTO 40
```

The uses of the AUDIO command include things like recording a sound track of questions for a question and answer program or giving instructions in spoken English about how a program should be used. Once you start thinking of times when AUDIO could be used all sorts of uses occur to you that other computers would have great difficulty in coping with. The only trouble is synchronisation. How do you make sure that the question program keeps in time with the questions being asked on the tape. The answer is of course the sound sense function!

Find a blank cassette and record the words "ONE", "TWO", "THREE" on it with a good silent gap between each. Then rewind it, press play and try the following program - (before typing this program in it is a good idea to switch the Dragon off and on to make sure that the sound sense line is set to input following the program in the last section that set it to an output.)

```
 10 DEF FNS(X)=1-(PEEK(&HFF22) AND 2)/2
 20 MOTOR ON
 30 AUDIO ON
 40 IF FNS(X)=0 THEN GOTO 40
 50 PRINT "ONE"
 60 IF FNS(X)=1 THEN GOTO 60
 70 IF FNS(X)=0 THEN GOTO 70
 80 PRINT "TWO"
 90 IF FNS(X)=1 THEN GOTO 90
100 IF FNS(X)=0 THEN GOTO 100
110 PRINT "THREE"
120 FOR I=1 TO 100
130 NEXT I
140 MOTOR OFF
```

If you adjust the volume control on the tape recorder correctly you should be able to get the words "ONE", "TWO" and "THREE" printed on the screen at the same time that they are said on the tape recorder. If you have trouble the problem is almost certainly that the silences between the words are not quiet enough. This is only a demonstration but you should be able to see that this same method could be used to test for the start of a recorded question on on a tape. In practice you have to use some care with the sound sense function to avoid 'triggering' it on the wrong sound but at least it is a step in the right direction!

BASIC sounds

The main theme of this chapter has been the flexibility of the Dragon's sound generating hardware. However, using BASIC there is not very much that can be done to increase the range of sounds that the Dragon can produce. Using the information presented here it is possible to make the Dragon produce all manner of sounds but only if you use assembler or something a lot faster than BASIC. Even though this is the case you can have a lot of fun creating simple tunes and sound effects using nothing but the BASIC commands SOUND and PLAY and this area is certainly far from fully explored. If you add the MOTOR and AUDIO commands to the list then the unexplored territory is even greater. Who needs a speech synthesiser when the Dragon can say anything you can!

Chapter Six

Advanced Graphics

Following Chapters 3 and 4 you might think that there is very little left to say about the Dragon's graphics - you would be wrong! The early chapters dealt mainly with the graphics hardware and the way that it interacted with and was controlled by Dragon BASIC. This still leaves the rather large subject of how to handle the software component of graphics for this chapter. Once again it is assumed that you have looked at the Dragon manual and know a little about the graphics commands PSET, PRESET, LINE, CIRCLE, DRAW, PAINT, GET and PUT. This chapter will be mainly concerned with how these commands can be used to achieve the objective of good graphics. The only trouble is that it is difficult to decide what 'good graphics' are!

Which of the commands you will find most useful depends very much on what you are trying to do and what sort of graphics interests you. As a result, the topics in this chapter are a collection of different ideas about how to use the Dragon rather than a single explanation of how things work. Although you are expected to know something of the graphics commands it is all too easy to miss interesting points when their syntax is complicated so you will also find some extra explanation of some of them.

Paged Graphics

One of the most confusing things that beginners have to get used to is the Dragon's bewildering range of commands that seem to do nothing but get you into a graphics mode. For example a high resolution graphics program might begin -

```
10 PCLEAR 2
20 PMODE 1,1
30 COLOR 4,1
40 SCREEN 1,0
50 PCLS 2
```

and all this just to obtain a clear yellow PMODE 1 screen! It is true that some of these commands are not needed in the majority of programs but they do exist and it is sometimes difficult for the beginner to sort out what does what. Although a brief description of the standard graphics initialisation commands was given in Chapter 4 it is worth going through the above program line by line. Line 10 reserves two 'pages' of memory to store the graphics information. These two pages are for graphics use and graphics use alone - BASIC etc has to keep out! Line 20 informs the Dragon that you want all subsequent graphics commands to use a PMODE 1 screen starting at the first page that you reserved using PCLEAR. At this point nothing new is displayed on the TV screen. You could use PSET, LINE or any other command to draw graphics information into the area of memory that you reserved but nothing would show on the TV set. Line 30 sets the default foreground and background colours to be used by any graphics commands that follow. You should realise by now that this command doesn't change anything already in the memory or on the screen. Line 40 is the first command in the program that actually changes what is displayed on the TV screen. This switches the display from the text screen or, as we shall see, the current graphics screen, to the one referred to by the last PMODE command. It also selects the colour set that will be used to display the screen. Finally, line 50 clears the screen to colour 2 or yellow. Notice that if PCLS had been used without a specified colour then the default background colour would have been used.

You should by now be used to the idea that the Dragon can draw on a graphics screen while displaying a text screen. What you probably are not so familiar with is the idea that the Dragon has a number of graphics screens that it can draw on and a number of screens that it can display! If you look at the memory map given at the end of Chapter 2 you will see that there are eight areas of memory called graphics page 1 to graphics page 8. Each one of these pages corresponds to 1.5K of memory and each high resolution graphics mode needs an exact number of pages of memory to store the data relating to its display

PMODE	pages needed
0	1
1	2
2	2
3	4
4	4

If you do not use a PCLEAR command within a program then BASIC prepares for the worst and sets aside four pages for graphics and reclaims the other four for its own use. If you know that you are going to use PMODES 0,1 or 2 then you can use PCLEAR to free the graphics pages that you are not going to use but unless the BASIC program that you are writing is a very big one this usually not essential.

What is more interesting is the prospect of using PCLEAR to reserve more graphics pages than the usual four. If you reserve all eight then you could have eight distinct screens in PMODE 0, four in PMODEs 1 and 2 and two in PMODEs 3 and 4. By a careful use of the PMODE and SCREEN command you could contrive to display one graphics screen while drawing another. Which screen you are drawing on is selected by PMODE m,s where m is specifies the graphics mode and s specifies the page number where the screen starts. So for example in PMODE 0 one of the eight screens would be selected by PMODE 0,I where I was the screen number. This is all very simple but there is a slight problem when it comes to PMODEs that use more than one graphics page per screen. The question is which page number do screens start on. For example in PMODE 1, which uses two graphics pages per screen you could first draw on the screen starting at page 1 (i.e. PMODE 1,1) and then draw on the screen starting at page 2 (i.e. PMODE 1,2). While this is perfectly good BASIC there are very few occasions where this would make sense. What you are doing is to make a screen up from pages 1 and 2 and then make another screen up from pages 2 and 3. Obviously with graphics page 2 in common these two screens are not independent, in fact the top half of one is the bottom half of the other! Apart from very exceptional cases, the only way to work is to divide the graphics pages up into non-overlapping screens and work only with these! That is -

PMODE	screen	pages
1,2	1	1 & 2
	2	3 & 4
	3	5 & 6
	4	7 & 7
3,4	1	1,2,3 & 4
	2	5,6,7 & 8

Notice that the 'screen' numbers are not recognised by BASIC but they do

make thinking about paged graphics a lot easier. To select any screen all you have to do is use the page number of its first page in the PMODE command.

Perhaps the most common use of paged graphics is in drawing one graphics screen while another is displayed. In fact this is a technique that most graphics programs would be improved by - it give a very bad impression if the user has to sit and watch a clever graphics screen being laboriously drawn. However there is one other interesting application for paged graphics - paged animation. If you draw a number of graphics screens, each one slightly different and then display them in sequence you can create the illusion of movement. The standard way of achieving this sort of animation is based on the draw/re-draw cycle. In other words you draw the picture, erase it and then re-draw it slightly altered. The trouble with the draw/re-draw method is that the fastest movement that you can achieve is limited by the time it takes to draw the picture. Usually this means that the picture is very simple and only a very small portion of it is erased and re drawn. This means that many animation tasks are out of the range of basic and you have to resort to assembler. For example, consider the simple problem of animating a number of circles or disks that 'pulsate'. The draw/re-draw method would require that all the circles were drawn with a small radius, then erased and drawn with a slightly bigger radius. This cycle would continue until the circles reached full size and then it would go into reverse, drawing the circles smaller each time and so on. Even though the Dragon is, very fast at drawing circles this simple method would be far too slow even with only two or three circles, but paged graphics allows you to animate as many pulsating circles as you like and even needs a delay loop to slow it down!

The basic idea behind paged animation is to draw a slightly different version of the picture on each of the screens available in the PMODE, and then display each one in turn. Notice that the drawing of the pictures and the actual animation are separated into two different stages. You can take as long as you like drawing each picture because the speed of animation is only affected by how long it takes to 'flip' between the different screens! As an example of paged animation, the following program implements the 'pulsating circle' idea described above -

```
10 PCLEAR 8
20 GOSUB 1000
30 FOR J=1 TO RND(50)
40 X=RND(255)
50 Y=RND(191)
60 GOSUB 2000
```

```
70 NEXT J
80 GOSUB 3000
90 GOSUB 4000
100 GOTO 80

1000 FOR I=1 TO 8
1010 PMODE 0,I
1020 PCLS
1030 NEXT I
1040 RETURN

2000 FOR I=1 TO 8
2010 PMODE 0,I
2020 CIRCLE(X,Y),I*2
2030 NEXT I
2040 RETURN

3000 FOR I=1 TO 8
3010 PMODE 0,I
3020 SCREEN 1,1
3030 FOR K=1 TO 50
3040 NEXT K
3050 NEXT I
3060 RETURN

4000 FOR I=8 TO 1 STEP -1
4010 PMODE 0,I
4020 SCREEN 1,1
4030 FOR K=1 TO 50
4040 NEXT K
4050 NEXT I
4060 RETURN
```

The main program is easy to understand. Line 10 reserves all eight graphics pages for use in PMODE 0 giving a total of eight different screens. Subroutine 1000 PCLSs each screen in turn. The FOR loop between lines 30 and 70 generates a random number of pulsating circles centered at X,Y. Subroutine 2000 is the part of the program that actually draws the pulsating circles. It does this by selecting each of the eight screens in turn and drawing a single circle on each. As the radius of the circle is I*2, where I is the page number, you should be able to see that the size of the circle increases as the page number increases. After drawing all the circles, the main program simply calls

subroutine s 3000 and 4000 repeatedly. Subroutine 3000 takes us though a display cycle where the circles increase and subroutine 4000 takes use though a display cycle where the circles decrease. They work by selecting a page to be displayed using PMODE 0,I and then displaying it using SCREEN 1,1. Notice that to make the animation run slow enough to be smooth and give the impression of pulsating, the display loop has to be slowed down by a delay loop! If you want to see how fast things can move delete the FOR loop on K in both subroutine 3000 and 4000.

If you would like to see some variations on this program try adding,

```
2025 PAINT (X,Y), 1,1
```

which produces pulsating disks. If you feel even more adventurous you might like to add line 2025 and

```
  55 D=RND(2)-1
2015 IF D=0 THEN R=(9-I)*2 ELSE R=I*2
2020 CIRCLE (X,Y),R
```

These last changes will produce a screen full of disks some of which increase in size while the rest decrease and vice versa! i leave further improvement to you but I cannot resist pointing out that the 'sound sense' subroutine described in the last chapter along with the AUDIO command could be used to synchronise the changes in the disks to whatever music might be be playing over the TV set - a Dragon disco is very possible!

The is one other command in Dragon BASIC that is concerned with graphics pages, PCOPY. This will copy the contents of any graphics page to any other. The only complication with using this command is remembering to copy all the pages that make up a screen in the correct order. For example, in PMODE 0 you can copy a whole screen to another screen using just one PCOPY command e.g. PCOPY 1,6 will copy the contents of the screen corresponding to graphics page 1 to the screen corresponding to graphics page 6. However to copy a PMODE 2 screen you would need two PCOPYs e.g. PCOPY 1,3:PCOPY 2,4 would copy the whole of screen 1 to screen 2. There may be applications where you only need to copy part of one screen into another but they are not common. One of the most useful applications of the PCOPY command is in 'background preservation'. Suppose you had written a program that played a game using small moving objects against a complicated background and as the game progressed the background graphics were altered. To play the game a second time would mean

re-drawing the complicated background. As an alternative you could draw the background into one screen and copy it into another screen where the game would actually be played. Renewing the background would now only take a few PCOPY commands!

The DRAW command

One criticism often levelled at the Dragon is that it doesn't have any user-defined characters. While this is quite true it does have the DRAW command which in many ways makes user-defined characters unnecessary! The usual way to introduce the DRAW command is to show how it can be used to draw large shapes, for example, a square would be "U40R40D40L40". There is no doubt that drawing large irregular shapes is a useful application of the DRAW command but it has another less obvious application in drawing small solid shapes. If you read the Dragon manual you will come to the conclusion that solid shapes are produced by first drawing a large outline and then PAINTing the interior of the shape. This is the best way to produce large shapes but the DRAW command alone is by far the better way to produce small solid shapes and small solid shapes are what other machines refer to as user-defined graphics.

The DRAW command is very easy to use and there is little point in repeating the details of the movement commands given in the Dragon manual. There are, however, a few points of detail that it is worth going over. The best way to think of the action of the DRAW command is to imagine a 'pen' that is moved, its colour changed etc, by the one letter commands within the draw string. Whenever you give the pen a movement command such as U5 or E4 it will move from its current position to its new position leaving a mark in its current colour. You can change the colour of the pen using the C command, lift it from the screen so that it leaves no mark by placing B in front of any command and move it to an absolute position on the screen using the Mx,y command. One of the most neglected draw commands is the move relative command. If you write M10,10 the pen will move to the point on the screen with co-ordinates 10,10 but if you write M+10,+10 the pen will move to x+10,y+10 where x,y is its current position. Similarly M-10,-10 will move the pen to x-10,y-10. Using this command it you can create relative movements like those produced by the commands U, D, etc but at an angle other than a multiple of 90 or 45 degrees.

Another command that is often misunderstood is the An or angle command. Depending on the value of n all of the lines produced by subsequent movement commands are rotated through 0, 90, 180 or 270 degrees. This sounds very clever but as all the movement commands are multiples of 90 or 45 degrees all that is amounts to is temporarily changing the meaning of the commands. For example, following A1 (rotate through 90 degrees) the U command behaves like the R command, the R command like the D command and so on. What is perhaps more important is the effect that these changes have on the outline that is drawn following such a rotation. An outline drawn following an An command will appear to have been rotated about the point that the pen was at when the An command was given. For example, try -

```
10 PMODE 0
20 SCREEN 1,1
30 PCLS
40 FOR I=0 TO 3
50 DRAW "BM100,100;A"+STR$(I)+"U50L10U5R10D5"
60 NEXT I
```

which will draw the same shape starting from the same location with four different rotations.

A command that is easy to understand but not used as often as it might be is the Sk or Scale command. Following an Sk command all drawing instructions have the same effect as if they had been written k*n/4 times as large. For example, following S8, U5 moves the pen 10 (=8*5/4) units up. The scale command is useful because it allows us to design a shape with a draw string working at a large scale and then when it is correct reduce the scale to give the size of desired. It also has an important use in designing the Dragon's equivalent of user-defined characters, which is discussed below.

The previous example introduced an idea which turns out to be the key to using the DRAW command more flexibly. If you look at line 50 you will see that the angle through which everything is rotated is set by the variable I which is added into the middle of the draw string each time the DRAW command is carried out. The most important thing to realise about the draw string is that it is a string, just like any other. This means that you can use all of the standard BASIC ways of manipulating strings to construct a draw string. For example, if A$ contains the commands to draw a shape but no commands

that move the pen to an absolute location (i.e no Mx,y commands) the shape can be drawn at any position by -

```
DRAW "BM"+STR$(X)+","+STR$(Y)+";"+A$
```

where the variables X and Y contain the co-ordinate of the position that the pen will start drawing the shape from. Notice the use of the STR$ function to convert the numbers stored in X and Y to strings. The draw string is indeed a string and any numbers that you want to include in it should always be converted into string form and then concatenated with the rest of the draw string in the correct position. Using this basic idea you can arrange for other features of a DRAW command to be changed by the values stored in a variable. For example,

```
DRAW "C"+STR$(N)+A$
```

will draw the shape specified by the commands in A$ in the colour specified by N (as long as A$ doesn't contain any colour commands of its own). Another way of achieving the same ends is -

```
DRAW C$+A$
```

with C$ set to Cn where n is the colour code desired, e.g. C$="C0".

This ability to specify colour and absolute position of a shape is so useful that unless there is a good reason to the contrary the follow guidelines are worth following -

- design the shape using a draw string without absolute movement and without any colour commands
- store the commands in a string variable (hopefully with an appropriate name)
- if you need to use more than one colour use a different variable for each colour component of the shape
- when you want to actually draw the shape set the position and colour information as follows:

```
X=x:Y=y:DRAW "CnBM"+STR$(X)+","+STR$(Y)+";"+s
```

where x,y is the position that you want the shape drawn at, n is the colour code for the colour that you want the shape drawn in and s is the string variable that contains the movement commands for the shape.

If you stick to these guidelines then you will find that your programs are easier to change and draw strings can be re-used to produce the same shape at different places and in different colours. A typical example of the using a draw string in this way is in the draw/re-draw method of animation described in the last section. Suppose we want to animate a single square outline across the screen this can be achieved by defining a draw string for the shape, drawing it in the foreground colour and then erasing it by drawing it in the background colour. Repeating this at a number of locations give the impression of movement. Try -

```
10 PMODE 4
20 PCLS
30 SCREEN 1,1
40 SQ$="U6R6D6L6"
50 Y=100
60 FOR X=0 TO 200
70 DRAW "C5BM"+STR$(X)+","+STR$(Y)+SQ$
80 FOR K=1 TO 10:NEXT K
90 DRAW "C0BM"+STR$(X)+","+STR$(Y)+SQ$
100 NEXT X
```

Line 40 sets up the square shape, line 70 draws it at X,Y in colour 5 and then line 90 removes it by re-drawing it in colour 0, the background colour.

Using the DRAW command to produce small solid shapes suitable for use in games is easy enough in theory, but there are a few practical pitfalls to be aware of. Suppose we want to define a small solid ball to animate as part of a game, we could use the CIRCLE command to draw a circle and then use the PAINT command to make it solid but this would be very slow. The pattern of dots that make up a small solid disk is might be something like

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

This solid shape can be produced by a DRAW command if the pen is moved to each dot in turn. It doesn't matter if some of the dots are visited more than once, this simply wastes a little time as long as the pen reaches every dot. For example, the pen could 'cover' the dots in the following way -

```
start *-> -*---*---*
   *-< -*---*---*---*---*
*---*---*---*-> -*---*---*---*
*---*---*---*-< -*---*---*---*
*---*---*---*-> *---*---*---*
   *---*---*-< -*---*---*
      *---*-> -*---*end
```

Giving rise to the following draw string -

```
R3F1L5G1R7D1L7D1R7G1L5F1R3
```

To see this draw string in action change line 40 in the moving square example given above to

```
40 SQ$ = "R3F1L5G1R7D1L7D1R7G1L5F1R3"
```

and you will see a small white ball move smoothly across the screen! In a games program you wouldn't move the ball just one location at a time you would probably choose something more like 6 to 8 locations per step to increase apparent speed. To see the effect of increasing the amount that the disk moves each time change line 60 to -

```
60 FOR X=0 TO 200 STEP 6
```

This idea of using DRAW to produce solid shapes seems easy enough but there is problem to beware of if you are using anything other than PMODE 4. As already explained in Chapter 5, in PMODE4 there areas many dots on the screen as there are distinct co-ordinates, that is no two co-ordinates refer to the same point on the screen. However, this is not the case in the other PMODEs and as demonstrated in Chapter 4 this can lead to some unexpected effects when using PSET, PRESET, and LINE. These unanticipated effects also crop up when using the DRAW command. For example, try the following program which draws a small 'dog' shape using the technique for solid shapes described above -

```
 10 PMODE 4
 20 PCLS
 30 DOG$ = "R1D1L1R1D3U2R3D2U3"
 40 SCREEN 1,1
 50 Y=100
 60 X=32
 70 GOSUB 1000
 80 X=47
 90 GOSUB 1000
100 GOTO 100
```

```
1000 DRAW "C1BM" + STR$(X) + "," + STR$(Y) + DOG$
1010 RETURN
```

Two copies of the dog shape are drawn, one at 32,100 and one at 47,100 and both look more or less the same. After running this program change line 10 to

```
10 PMODE 0
```

and re run the program. The results are rather surprising. Not only does the dog shape look different, the two copies are no longer the same! The reason for this is quite easy to see once the effect has been pointed out. In PMODE 0 there are only half the number of dots in the x and y directions than the maximum x an y co-ordinates would lead us to believe. This means that sometimes a move of one co-ordinate position in a draw string takes us to a new screen point and sometimes it does not. The first effect that this has is that some of the fine detail in a draw string may be lost, changing the shape that you are trying to produce on the screen. The second effect is that a different part of the fine detail will be lost depending on whether the shape is started from an odd or even co-ordinate location. So, the dog drawn starting at 32,100 looks different from the dog drawn at 47,100. If you try to animate the dog the effect looks even worse as the shape changes as the dog moves across the screen. This loss of fine detail can occur even on shapes that are large compared to the dog. Indeed in PMODE0 or 1 any draw string moves of one can lead to trouble!

The answer to the problem is to avoid trying to use more resolution in a draw string than is available on in the PMODE that you are using. One way to do this is to design any shapes that you want to use in PMODE 4 where no strange things can happen and then use the resulting draw string in PMODE 0 or 1 but with a scale factor of 8. To see this, change the dog's draw string to include S8 at the start in the previous program. In PMODEs 2 and 3 things are a little more complicated because you can use the full y resolution but not the full x resolution. That is, draw strings may contain moves of 1 unit in the y direction but only moves of 2 units in the x direction will always give you a new point on the screen. This seems simple enough until you ask what is the smallest diagonal movement that always guarantees a new point. (The answer is one!)

Whenever you use DRAW or any of the other graphics commands you should always ask yourself whether you are trying to use more resolution than the screen allows. As we have seen, the effect of doing this is not just the loss of fine detail, it can cause moving objects to change and shimmer in a very

irritating manner. It may be this misunderstanding that has lead to the Dragon's graphics quality being criticised when things are moving on the screen and the claim made by some programmers that PMODE 4 gives better results!

Before leaving the topic of DRAW it is worth pointing out that alphanumeric characters are just a special case of small solid objects and the techniques described above can be used to give both upper and lower case characters to Dragon high resolution graphics.

GET and PUT: user-defined characters

Of all the graphics commands on the Dragon, GET and PUT seem to be the most powerful and the most mysterious. Using GET you can save any rectangular area of the screen in an array and can then use PUT to produce new copies anywhere on the screen. In practice the need for these two commands is less than you might expect. It is often easier and faster to re-draw the shape then to use GET and PUT to reproduce it. However, a little knowledge of how the two commands actually do their jobs leads us on to ways of using them that the Dragon manual never hints at.

What happens when you use GET depends on whether you end the command with 'G' or not. If you use the GET command without 'G' then all that happens is that the display information within the specified rectangle is read into the array memory location by memory location. In other words, GET transfers whole screen memory locations into the array. For example if you GET a 6 by 6 square into the array A in PMODE 4 using

```
GET (0,0)-(5,5),A
```

then six memory locations within the array will be used. The first will store the screen memory location that holds the to row of six dots within the rectangle, the second will store the screen location that holds the second row of dots and so on down to the sixth row of dots. Now in PMODE 4, or in any two colour mode, a screen memory location determines the colour of eight dots so although you might think that the GET instruction will only store the information about the six rows of six dots within the rectange you actually store information on six rows of EIGHT dots whether you like it or not!

This idea of saving whole screen memory locations is quite a tricky one so it is worth looking at the way the screen memory locations affect what is displayed on the screen. Although the memory maps for the different graphics modes have been discussed in Chapter 4 their meaning might not be clear enough for you to follow what is happening with GET. In a two colour mode, each memory location within the screen area determines the colour of eight dots. In the same way, in four colour mode each memory location controls that colour of four dots on the screen. What the memory map equation in Chapter4 does is to take the co-ordinates of any dot on the screen and give you the address of the memory location that determines its colour. While this is useful, it doesn't really help you to imagine how the memory map corresponds to screen locations. If you imagine the screen as made up of horizontal rows of dots then, in a two colour mode, the first eight dots in the top row are controlled by the first screen memory location, the next eight to their right by the second memory location and so on to the end of the row. In a four colour mode the correspondence is the same but with groups of four dots rather than eight. When you move down to the screen to the second row of dots the pattern is repeated, and so on to the very bottom of the screen. To see this process in action in a two colour mode try the following program

```
 10 PMODE4
 20 PCLS
 30 SCREEN 1,1
 40 A=&HFF
 50 B=0
 60 I=0
 70 FOR X= &H600 TO &H600+4*&H600 STEP 2
 80 I=I+1
 90 POKE X,A:POKE X+1,B
100 FOR K=1 TO 100:NEXT K
110 IF I=16 THEN I=A:A=B:B=I:I=0
120 NEXT X
130 GOTO 130
```

What this program does is to POKE all ones or all zeros into alternate screen locations so that you can see the group of eight dots that each controls. You should be able to see that the screen is divided into 32 columns, each eight dots wide. If you want to see the pattern for a four colour screen then change line 10 in the last program to read 10 PMODE3. You will seethe same pattern, only now each of the lines contains only four screen dots.

These groups of eight or four dots that are controlled by a single memory location are very important to the understanding of the way GET and PUT work. If a screen memory location controls even one of the dots within the GET rectangle then the whole memory location is saved in the array and this means that all the information about the other points that it controls is saved in the array. In this sense you can imagine the rectangle as a sort of frame that is placed on the screen and any screen memory location that is even partially within it is stored in the array.

The screen locations are stored in the array in row order in the same way that the screen memory locations correspond to screen positions. That is, the memory locations from the first row within the rectangle are stored in the array first, working from left to right, then the second row and so on. If you have used GET to store graphics information in an array, you will have found that when you try to examine the contents of the array you find mainly zeros. The reason for this is that GET uses the array in a way that completely ignores the way the array, or even standard BASIC numbers, are stored. The details of how BASIC stores arrays aren't dealt with until Chapter 8 and all you need to know now that when you dimension an array BASIC sets aside an area of memory large enough to store all the elements of the array. So, for example, DIM A(25) reserves 26*5 memory locations because the array has 26 elements and each element (a real number) needs 5 memory locations. Similarly DIM A(9,9) reserves 100*5 memory locations. As far as the GET and PUT commands are concerned, the BASIC arrays are used as areas of storage. No use is made of the fact that the array is one- or two- dimensional. So, as far as GET and PUT are concerned, DIM A(1,4) and DIM A(9) provide the same amount of storage (i.e. 50 memory locations) and the screen memory locations are simply stored one after the other in this area of storage.

Now although the Dragon manual suggests that if you want to GET a rectangle with n*m points in it, you need an array DIM A(n,m) this is in fact a gross over-provision of memory. It will always work but it is always too much! If you want to GET a screen rectangle n*m into the smallest array that will hold it all you have to do is to work out how many screen memory locations are within the rectangle and dimension an array that reserves this many memory locations. This is, of course, not an easy calculation but even an approximation saves a lot of memory. For example if you are GETting a 5 by 5 rectangle in PMODE 4 the largest number of memory screen memory locations that this involves is ten (i.e. two screen memory locations per row) so the array can be as small as DIM A(1) i.e an array with two elements. If you

follow the Dragon manual's recommendation you would have to reserve an array DIM A(5,5) i.e. 25 array elements or 25*5 memory locations!

A consequence of GET storing whole screen memory locations is that PUT restores whole memory locations. This means that if you GET a rectangle that includes part of a screen location, when you PUT it back anywhere else on the screen all of the memory locations that were stored in the array are restored to the screen and so an area outside of the rectangle that you specified in the PUT is altered. What is worse, PUT restores the screen memory locations in the same way as GET stored them and this means that if the PUT rectangle happens to overlap fewer or more screen memory locations than the GET rectangle did the result will be a confused mess! This fact is usually ignored in descriptions of how GET and PUT work together but if you want to check that it is true, try -

```
10 PMODE 4
20 PCLS
30 DIM A(10)
40 SCREEN 1,1
50 LINE (0,0)-(7,5),PSET,BF
60 GET (7,0)-(12,5),A
70 PUT (12,50)-(17,55),A
80 GOTO 80
```

If you try to predict what this program will do you might be surprised when you run it. Line 50 draws a small solid square with corners at 0,0 and 7,5. The GET statement should store screen information in A from within a square with corners at 7,0 and 12,5. This GET square only overlaps the solid square in a small vertical strip one dot wide, i.e. the line from 7,0 to 7,5. (If you are confused try drawing things out on graph paper.) When you PUT the information back on the screen all you would expect to see is this vertical strip. What you in fact see is the whole of the small square reproduced at the new location! The reason for this is, of course, that the GET doesn't store information just from within the area specified, it stores whole screen memory locations. If you think about the memory map for PMODE 4 you will realise that the GET area just overlaps two columns of adjacent screen locations. For example, the first screen memory location determines the colour of the first eight dots in the top left hand corner of the screen and the GET area includes the eight dots on the top line, so this whole screen memory location has to be stored along with screen location two, which determines the colour of the next eight dots that also fall within the square. The situation can best be summed up by Fig 6.1,

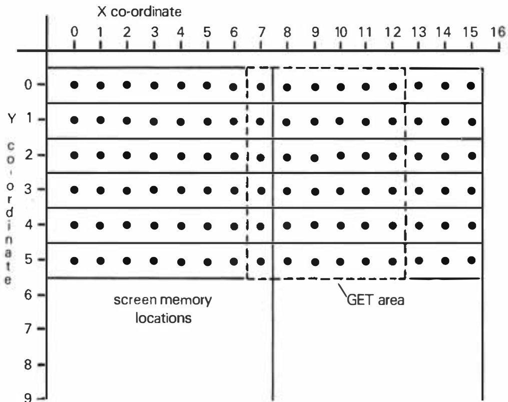


Fig 6.1 The action of GET

from which you can see that although the 'GET area' is small, it in fact GETs all the 12 memory screen memory locations that it overlaps. (Remember that in a two colour mode a screen location determines the colour of a single row of eight dots.)

To overcome this problem there is a second form of both the GET and the PUT command. If you follow the GET command with a G then, instead of storing whole memory locations, each dot on the screen is treated separately and the bit or bits that control it are 'packed' into the array. The original groupings into eight or four dots controlled by the same screen memory location are completely ignored. This means that the only graphics data that is stored in the array comes from within the area defined by the GET rectangle and the PUT command only alters the area of the screen within it's specified rectangle. To see this add G to the end of the GET command in the last

example and PSET to the end of the PUT command and you will see that instead of a square appearing at the new location all you get is the vertical line that was predicted! All in all it is easier to predict the result of a GET command ending in G than the result of a GET command that doesn't! For this reason it is better to always to use the G unless you have a good reason for not doing so.

To summarise -

The GET command works by storing whole screen memory locations into an array row order and from left to right. The PUT command simply transfers these memory locations back to the screen wherever they are required. If you follow the GET command with a G then screen memory locations are ignored and the screen information is stored in the array dot by dot and no information outside the GET rectangle is stored. In this case the PUT command will not affect anything outside it's specified rectangle.

As a side effect the GET and PUT commands are slower when G is specified but you can use additional commands in PUT, such as PSET, NOT etc to determine how the information will be restored to the screen.

All this is a little complicated to follow as it requires a good understanding of the format that is used to store screen data. If you want to examine the contents of an array that has been used in a GET to check that you understand how the information is stored, use -

```
1000 FOR I= VARPTR(A(0)) TO VARPTR(A(d)) + 5
1010 PRINT PEEK(I);",";
1020 NEXT I
```

where d is the maximum dimension of the array A (i.e. DIM A(d)). This routine will dump the contents of each screen location that forms the array. If you want to know how it works then look up the definition of VARPTR in the Dragon manual. More information about the format of arrays and other data storage is given in Chapter 8.

Apart from explaining the odd things that can happen if you use GET without a G at the end, knowing how the GET and PUT commands work seems a little academic. However, now that we know the data format used by the simple GET command we can store data directly into an array to define the shape that would be produced by a PUT. In other words, we can PUT an array that contains graphics data that has never been the subject of a GET

command. This opens the door to another way of producing user-defined characters.

The easiest way of working is with the GET command without the final G and hence in terms of complete screen memory locations. The reason for this is that it is very easy to define the bit pattern of a single memory location and then use PUT to transfer this bit pattern to a screen memory location and thus to a pattern of dots. To avoid problem with PUT rectangles overlapping different numbers of screen memory locations as it is moved about the screen we also have to restrict ourselves to placing the PUT rectangle over whole screen memory locations. This corresponds to making sure that the top left hand corner of the rectangle is always at an x co-ordinate that is exactly divisible by 8. For example, if we want to define the disk shape used in the DRAW example given earlier we need to work with the screen divided into 8 x 8 squares. (The disk has eight rows and a maximum of eight dots in a row.) A program to both define the disk and 'bounce' it around the screen it given below -

```
 10 PMODE4
 20 PCLS 0
 30 SCREEN 1,1
 40 DIM A(7),B(7)
 50 V=VARPTR(A(0))
 60 POKE V,&H3C
 70 POKE V+1,&H7E
 80 POKE V+2,&HFF
 90 POKE V+3,&HFF
100 POKE V+4,&HFF
110 POKE V+5,&HFF
120 POKE V+6,&H7E
130 POKE V+7,&H3C
140 V=8
150 W=8
160 X=X+V
170 Y=Y+W
180 PUT (X,Y)-(X+7,Y+7),A
190 IF X>=247 OR X<=0 THEN V=-V
200 IF Y>=183 OR Y<=0 THEN W=-W
210 PUT (X,Y)-(X+7,Y+7),B
220 GOTO 160
```

Lines 50 to 130 define the dot pattern of the disk by storing the bit pattern for each row into the memory locations reserved by the array. If you convert each hex number to binary you will see that each one defines the dot pattern in a row of the disk (i.e. 0 gives a black dot and 1 gives a white dot.) The bit patterns are stored in successive memory locations within the array A, the first location of which is found using the VARPTR function. When A is PUT to the screen, each of its memory locations are transferred to the screen in turn to produce the image of the disk. If you want to see why the PUT rectangle has to be placed exactly over a block of eight screen locations change line 140 to 140 V = 7 and watch the chaos that follows.

Even if you haven't followed all of the details of how GET and PUT work you can still use the above method of defining characters directly. It is very fast and particularly useful for converting programs from machines that do have user-defined graphics.

Joysticks and lightpens

One of the first accessories that most people buy to use with the Dragon is a pair of joysticks. These are very easy to use as Dragon BASIC includes the JOYSTK function to read their position. However, there-is no built in function to read the state of the fire buttons. This omission can easily be remedied. The following user-defined function will test the state of the fire buttons

```
DEF FNB(S) = (PEEK(&HFF00) AND S)/S
```

FNB(1) changes from 1 to 0 when the fire button on the left joystick if pressed and FNB(2) changes similarly for the right joysticks button.

Using the joysticks to control the position of a point on the high resolution screen is simply a matter of reading the joysticks and using PSET and PRESET to plot and unplot points. Try -

```
10 PMODE4
20 PCLS
30 SCREEN 1,1
40 X=JOYSTK(0)*4
50 Y=JOYSTK(1)*3
60 PSET(X,Y)
70 PRESET(X,Y)
80 GOTO40
```

Notice that it is necessary to scale the inputs from the joysticks to make sure that you can reach all parts of the screen. To make the joysticks less sensitive to slight random movements it is a good idea to read each one a few times and take the average. There is a simple way of doing this that combines this averaging with the scaling procedure. Change line 50 and 60 to -

```
50 X = JOYSTK(0) + JOYSTK(0) + JOYSTK(0) +JOYSTK(0)
60 Y = JOYSTK(1) + JOYSTK(1) + JOYSTK(1)
```

If you want to draw on the screen then remove line 70. You may be disappointed to discover that the lines that you draw are rather dotted! The trouble is that the joysticks can be moved so fast that the plotted point jumps around the screen without 'visiting' any of the points in between. The solution to this problem is to draw continuous lines between each of the plotted points using the DRAW command. Change line 60 to -

```
60 DRAW "M" + STR$(X) + "," + STR$(Y)
```

and your sketches will look a lot better.

For many applications, a light pen is a better input device than a joystick. Although there are light pens on the market for the Dragon there is a fundamental limitation built into its hardware that limits that way that such pens can be used. Most other computer systems that use light pens keep an internal count of which line on the screen is being displayed. If you recall how a TV picture is built up by a scanning bright dot, then you will be able to see that at any one moment in time there is only one point on the TV screen that is very bright. (We see a TV picture only because our eyes are not fast enough to follow the scan.) If you point a light pen at a TV screen then it will pick up a pulse when the scanning spot passes beneath it. By looking at the internal count of the line being currently displayed when the pulse is detected the computer can work out where on the screen the light pen it. The trouble is that the Dragon doesn't keep a count of the line that is currently being displayed anywhere that the user can get at it so this traditional sort of light pen will not work.

What you can do is to connect a light sensor to the joystick input (how will be explained in the next chapter) and use this to detect the light level on the TV screen. To know where the light pen is on the screen you have to set up areas that are flashed on and off at given times. If the light pen is pointing at one of these areas then you can pick up the variation in the light input when the area is flashed. In this way you can work out which area the pen is pointing

at. This sort of light pen is very good for picking one of a number of options from a menu but not for drawing freehand on the screen.

What the Dragon lacks

At this point it is perhaps appropiate to consider what the Dragon can and cannot do as regards graphics. The simple answer is that if you are happy with two or four colours the Dragon is an excellent machine. What is lacking are a few simple software features that would make the programmer's life so much easier. For example, the high resolution graphics modes are so much easier to use than the semi-graphics and text modes that the Dragon would almost be better off without semi-graphics and text! It is perfectly possible to imagine a Dragon that uses the high resolution modes to list programs, to print out on and to accept input. Some extra software to provide user-defined characters and to print them on the screen would complete the picture of a more versatile Dragon. The strange thing is that this is all possible without a single hardware modification!

Chapter Seven

Interfacing

The subject of interfacing is a very wide one which covers connecting almost any piece of electronic equipment to any other. However, in this chapter our examination of interfacing will be limited to the ways in which the Dragon can be connected to the outside world. The Dragon has a number of such connections and the way that they work and what they do is quite diverse. However, apart from the expansion port they all make some use of a 6821 PIA and so this chapter starts with a discussion of this particular component. Then, the way that the two PIAs within the Dragon are used is explained. Some of these uses have already been described in previous chapters but there remain a number still to be discussed, including -

The keyboard, the printer port, the joystick inputs, and the cassette interface.

Finally the details of the expansion port are given.

Many of the technical details given in this chapter will also help you to understand features of the Dragon involving the PIAs that have been already described. Unless you have missed the number of times that the PIA has cropped up so far, you will not need convincing that it is a chip worth studying!

The 6821 PIA

A single 6821 PIA provides sixteen general purpose lines that can be used for either input or output and four special control lines. As has already been described in Chapter 2, these lines are grouped into two lots of ten - the A side and the B side. The A side is composed of eight general purpose input/output

lines usually called PA0 to PA7 and two of the special purpose control lines usually called CA1 and CA2. Similarly, the B side consists of the remaining eight general purpose lines and the two remaining control lines called PB0 to PB7 and CB1 and CB2 respectively. (See Fig 2.6 in Chapter 2.) The way that the control lines CA1, CA2, CB1 and CB2 can be used is the most sophisticated and complicated aspect of the PIA. Fortunately as far as the Dragon is concerned much of this complication can be ignored because the control lines are used in fairly simple ways. Likewise the fact the any of the sixteen lines PA0 to PA7 and PB0 to PB7 can either be used as an input or an output is largely academic as in most cases the Dragon sets them to one or the other when it is first switched on. The final point to make about the PIA is that all its lines, input, output or control, work with 0 and 5 volts. In other words an output line can be in one of two states: low at 0 volts or high at 5 volts. Likewise an input line, if properly used, should only have 0 volts or 5 volts applied to it. Anything higher or lower is likely to damage the chip and anything in between will give unreliable results.

From the programmer's point of view a PIA looks like four memory locations as shown below -

s+3	B control register
s+2	B data register
s+1	A control register
s	A data register

As you can see, the first two locations are concerned with the A side and the last two the B side. The A side data register is best thought of as a normal memory location which, in addition, is connected to the lines PA0 to PA7. Each of the eight bits of the data register bit0 to bit 7 is associated with the line of the same number. That is, bit 0 is associated with PB0, bit 1 is associated with PB1 and so on. The effect of this association depends on whether the line is used as an input or as an output. For an input line the state, i.e. 0 or 1, of the corresponding bit in the data register depends on the voltage applied to the line. If the voltage is high (5 volts) then the bit is a 1, otherwise, if the voltage is low (0 volts) then the bit is a 0. Thus by reading the data register you can find out the voltage applied to any input lines by examining the corresponding bits. For an output line the state of the line is also determined by the state of the corresponding bit. In other words, if the bit is0 then the line

is low at 0 volts and if the bit is 1 then the line is high at 5 volts. Thus by writing to the data register you can set the state of an output line.

In this description of the way that the data register you may have noticed the way that the PIA's 0 and 5 voltsseemedto pairnaturallywith the 0 and 1 of binary. This is not an accident, just a reflection of the way all computer hardware uses two voltage levels to represent the two states corresponding to 0 and 1.

The two control registers also behave like normal memory locations, only in this case the state of each of the eight bits controls some aspect of the way that the PIA works. The format of the two control registers is roughly the same and can be seen below -

b7	b6	b5	b4	b3	b2	b1	b0
IRQ 1 CA/B 1	IRQ 2 CA/B 2	CA/B2 control			DDR	CA/B 1 control	

Bits 3,4 and 5 control the way that either CA2 or CB2 operates. There are sixteen different operating modes possible for CA2 but fortunately the Dragon always uses CA2 and CB2 as outputs in the simplest possible way. (If you want to know more about the ways that CA2 and CB2 can be used then get hold of a full6821 data sheet.) In thismodebits4 and5 aresetto 1 and bit3 acts as the controlling bit for CA2 or CB2 as an output line - that is, if bit 3 is 1 the line is high and if it is 0 the line is low. As far as the Dragon is concerned this is theonlywaythatwecan use the CA2 and CB2 lines. Bits0 and 1 control the way that the CA1 and CB 1 lines work and in this case the Dragon allows us a little more freedom. The CA1 and CB1 lines are always input lines but they are different from any other input lines that we have looked at because they are 'edge triggered'. What this means is that instead of responding to the voltage applied to them they respond to a change in the voltage applied to them. If the voltage changes from 0 to 5 volts, this is called a 'rising edge' and if the voltage changes from 5 to 0 volts this is called a 'falling edge'. Depending on the setting of bit 1, the CA1 and CB1 inputs can be made sensitive to rising or falling edges -

bit 1	*triggered by*
0	falling edge
1	rising edge

When one of the inputs detects a voltage transition that it has been set to respond to, all that happens is that bit 7 in the control register is set. So if you look at bit 7 you can tell if the input line has detected either a rising or a falling edge. This leads to the question, how is bit 7 reset to 0. Bit7 is set to 1 when the the input is triggered but it is NOT set to 0 by a subsequent triggering, nor is it set to 0 by a voltage transition in the opposite direction. Once bit 7 is set to 1 by the input line it remains set to 1 until the associated data register is read. This may seem like an odd way of setting a bit to 0 but it makes very good sense when you consider the sorts of applications that CA1 or CB1 are used for. For example, if you wanted to use the A side data lines PA0 to PA7 as inputs from another computer you could use CA1 as a signal that the data was ready to be read in. Changing the voltage on CA1 would set bit 7 in the A control register and this would be taken as a signal that the data was ready on the input lines PA0 to PA7. Reading the A data register would then simultaneously get the data into the computer and clear bit 7 ready to be set by the next triggering on CA1, indicating that there was some more data to be read in.

Bit0 controls whether or not the setting of bit 7 causes an interrupt or not. Interrupts are something that most BASIC programmers never meet because they are more to do with the workings of the hardware than of high level languages. However, the idea behind an interrupt is not difficult to understand. Normally the 6809 CPU inside the Dragon is busy doing something like running your BASIC program or dealing with the letters that you are typing at the keyboard. When it receives an interrupt it stops what it is doing, saves enough information so that it can pick up where it left off and goes to do something else for a while. When it has finished this other job it returns to the original task. More technically, stopping what it is doing to start the other job is known as 'servicing the interrupt', the other job is called the 'interrupt routine' and returning to the original job is called 'returning from the interrupt'. The Dragon makes use of interrupts to provide the BASIC function TIMER, but more of this later. In general, interrupts are easy to understand but quite difficult to work with. For one thing you need to use assembly language and for another they are usually very involved with the way the machine works. Interrupts are disabled if bit 0 is 0 and enabled if bit 0 is 1. Unless you know what you are doing interrupts are best left for the Dragon to sort out!

There are only two bits left to explain in the control register, bit 6 and bit 3. Bit 6 works with CA2 and CB2 when they are set up as inputs in much the same way that bit 7 works with CA1 and CB1. However, as has already been

pointed out, the Dragon always uses CA2 and CB2 as outputs and so bit 6 is never of any use! Bit 3 gives access to another pair of registers within the PIA that are normally hidden from view - the data direction registers. These occupy the same addresses as the A and B data registers so you can either get at the data registers or the data direction registers but not both. It is therefore fortunate that the need to access them at the same time never arises because the data direction registers are used to determine whether the data lines are inputs or outputs. The way that this works is that each bit in the data direction register corresponds to an output line - i.e. bit 0 corresponds to PA0 etc. A line is an output if its bit in the data direction register is 1 and an input if it is 0. Thus by setting and resetting bits in the data direction register, a PIA can be made to have any combination of inputs and outputs that we need. Of course, the Dragon has already decided whether a line is an input or an output in most cases and so the data direction register doesn't usually concern us. But there are exceptions - see the section dealing with single bit sound in Chapter 5 for an example where this is not the case. If bit 3 is 1 then the A or B data register is selected and if it is 0 the data direction register is selected.

This is all there is to know about the 6821 PIA as far as the Dragon is concerned. In fact the only thing that has been left out in this description is the way that the CA2 and CB2 lines can be used as inputs (because the Dragon always uses them as outputs). If you want to know everything that there is to know about the PIA then this missing information should be easy to follow from a 6821 data sheet once you have read the rest of this chapter. Now it is time to move on from the theory of how the PIA works to the practical aspects of the way the Dragon uses it.

The Dragon's PIAs

The Dragon uses two PIAs to control not only a wide variety of input/output devices but also various aspects of its own working! We have already discovered some of the uses of the PIA output and input lines in earlier chapters and now it is time to give a complete list (see table 7.1) -

You should be able to see from this table that the PIA lines can be grouped according to what they do. The only line that will not be described any further is PIA 1's PA2 which is used to set the SAM chip to either 16K or 32K of RAM. As the Dragon always has 32K of RAM this line is somewhat redundant!

Table 7.1

PIA 0

FF00	A side data register
FF01	A side control register
FF02	B side data register
FF03	B side control register

A side	I/O	function	described/ used in Chapter
PA0to PA6	I	Keyboard row input	7
(PA0 & PA1)	I	also used for fire button input	6,7
PA7	I	Joystick A to D comparator input	7
CA1	I	TV line sync pulses	7
CA2	O	Sound source-joy stick selectb0	5,7
B side			
PB0 to PB7	O	Keyboard column output and printer interface	7
CB1	I	TV frame sync pulses	7
CB2	O	Sound source-joystick select bit 1	5,7

PIA 1

FF20 A	side data register
FF21 A	side control register
FF22 B	side data register
FF23 B	side control register

(continued......)

A side	**I/O**	**function**	**described/ used in Chapter**
PA0	I	Cassette input data	7
PA1	O	Printer strobe output	7
PA2 to PA7	O	Sound generator (D to A)	5
CA1	I	Printer acknowledge	7
CA2	O	Cassette motor control	7
B side			
PB0	I	Printer busy signal	7
PB1	I/O	single bit sound/sound sense	5
PB2	I	Ram size 16K/32K select	7
PB3	O	Colour set select to video gen.	3,4
PB4	O	GM0 to video gen	3,4
PB5	O	GM1 to video gen	3,4
PB6	O	GM2 to video gen	3,4
PB7	O	Alpha/Graphics to video gen	3,4
CB1	I	Cartridge detect	7
CB2	O	Sound enable to TV5	

Setting PIAs - the use of AND, OR and NOT.

Now that we know what each bit in the PIA control and data registers is for, we can begin to experiment and alter them. In assembly language and machine code this altering of individual bits is usually called 'bit manipulation' and there are special instructions for just this purpose. However, BASIC doesn't even recognise that binary numbers exist let alone individual bits! This is not a criticism of BASIC as it was never intended to be used for such advanced applications - after all it was originally a teaching language. For a beginner, the great advantage of BASIC is that it keeps bits and binary

numbers well hidden. The question is how can we go about altering individual bits using BASIC. Fortunately, Dragon BASIC does provide sufficient facilities to make this possible. In particular the logical operators AND, OR and NOT, apart from being useful in IF statements, can be used for bit manipulation. But first it is worth examining in detail the way that a hex number can be converted to binary and vice versa.

Hexadecimal number were briefly introduced in Chapter 2 as a way of specifying memory locations. One of the reasons for using hex numbers is that they are very easy to convert to binary and vice versa. As the Dragon will accept hex numbers and not binary numbers it is useful to be able to convert from one to the other using some simple rules -

Hex to binary

Each hexadecimal digit corresponds to four binary bits as follows -

Hex	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Given a hex number, look up each digit in the table and write down the four bits in its place. For example, E2A4 in hex is -

E	2	A	4
1110	0010	1010	0100

Binary to Hex -

To convert a binary number to hex just divide the binary number into groups of four starting from the righthand side. If the final group of bits is less than four then add zeros to the left to make up the number then look up the hex digit that each group corresponds to in the table given above and write them down in the same order. For example, 1010011 is -

|0101| |0011|
5 3

Notice that as one hex digit corresponds to four binary bits it only takes a two digit hex number to specify all eight bits in a memory location.

Now that we know how to convert between binary and hex we can look at the problem of bit manipulation. Suppose, for example, that we want to set bit3 to a 1 in a PIA data register. You might think that all we have to do is store (POKE) 00001000 (remember we number bits starting from bit 0) in the register but this would not only set bit3 to 1 it would set all the other bits to 0! Obviously we have to find some way of changing bit 3 without altering any of the other bits. The answer is to use the BASIC operator OR. This can be used to combine two numbers together rather like addition or any other arithmetic operation. The result of A OR B is obtained by taking each pair of bits one from A and one from B and working out a single bit of the result using -

A	**B**	**A OR B**
0	0	0
0	1	1
1	0	1
1	1	1

For example, if A is 0101 and B is 1001 the result of A OR B is 1101. In other words, there is a 1 in the answer if there was a 1 in the same place in either A or in B. Using this idea you should be able to see that to set bit 3 in the data register without altering any other bits all we have to to is read what is already in the data register and then OR it with 0000 1000. The result will have a 1 wherever the original contents of the data register did and a 1 in bit3 no matter what was already there. The final BASIC command to set bit 3 to 1 is -

```
D = PEEK(s) OR &H08  POKE s,D
```

where the first line works out the value that has to be stored in the data register whose address is s and then the next line actually stores it there. Notice the use of the hex equivalent of 00001000 in the first line.

Similarly to set a bit to 0 without altering any of the other bits we have to use the AND operation. This works in the same way as OR except that - there is a 1 in the answer only if there was a 1 in the same place in both A and B.

So, for example, 0101 AND 1001 is 0001. Now suppose we want to set bit 3 to 0 this can be done by ANDing 11110111 with the current contents of the data register. Bit 3 will be set to 0 no matter what its current value is but the other bits will not change. The resulting BASIC is -

```
D= PEEK(s) AND &HF7
POKE s,D
```

If you look at the above example you will notice that 1111 0111 is the same as 00001000 but with 0 written for 1 and 1 written for 0. You can make this change automatically using the BASIC operator NOT. The result of NOT A is simply obtained by writing 0 for 1 and 1 for 0 for each bit of A. For example, NOT 10100 is 01011. The example of setting bit 3 to 0 can now be written -

```
D= PEEK(s) AND NOT(&H08)
POKE s,D
```

To generalise these ideas, you can set any group of bits to 0 or 1 in exactly the same way. For example, to set bit 7, bit 3 and bit 0 to 1 all you have to do is OR the original value with 10001001 or &H89. To set the same bits to 0 you would AND the original value with NOT(&H89). Once you get used to these functions, bit manipulation from Dragon BASIC is easy.

There are two other simple operations that are worth knowing about. If you multiply a binary number by two this is equivalent to moving all the bits one place to the left and adding a zero to the right. For example, 10110 * 2 is 101100. In the same way dividing by two is the same as moving all the bits to the right and 'chopping' off the rightmost bit. For example, 10110/2 is 1011. In other words, multiplying by two is a 'shift left' operation on a bit pattern an dividing by two is a 'shift right' operation on a bit pattern. You will find many examples of the use of AND, OR, NOT, multiplying by two and dividing by two in earlier chapters of this book and a few more are yet to come. If you find the ideas of bit manipulation difficult on the one hand or interesting on the other then the best advice I can give is to read something on the theory of binary numbers. Now why you know that such things are important you should find the theory relevant and useful.

The keyboard

The Dragon's keyboard is a very simple affair from the point of view of electronics, (see Fig 7.1) The keys are arranged so that pressing one will

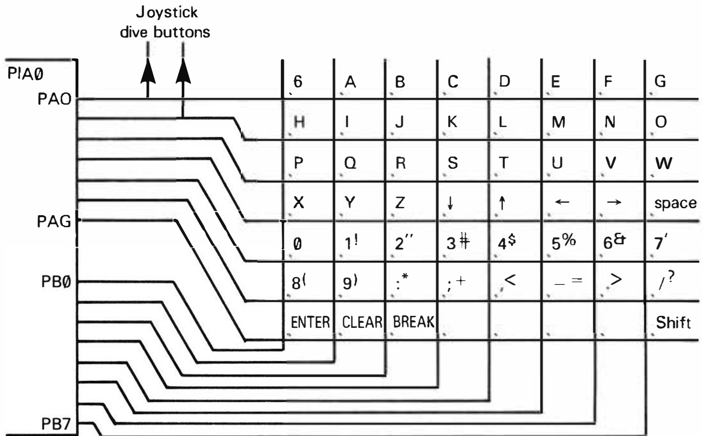


Fig 7.1 The Keyboard

connect one of the eight output lines from PIA 0 to one of the seven input lines. The keyboard is read by looking at each column of keys in turn. The output line of the column that is being examined is set to low and all the other lines is set high. If a key is pressed in that column then the input line corresponding to the row that the key is in will be connected to the low column line. This will cause a zero to appear in the input line's data register. All the other lines are high because either they are not connected to any output lines (if an input line is not connected it 'floats high') or they are connected to a high output line. Thus, if you set a column line low and read the input line's data register, any zeros correspond to keys that have been pressed in that column. By scanning through each column output line one-by-one the whole keyboard can be read.

This scanning and reading is carried out by software in the BASIC ROMs but, if we want to, the keyboard can be read directly from BASIC. For example, if you want to read the keys on column 0 all you have to do is set bit 0 of the B side data register to 0 and the rest to 1 and then read the A side data register. Try -

```
10 POKE &HFF02,&HFE
20 PRINT HEX$(&HFF00)
30 GOTO20
```

You will find that the number printed out changes as you press different keys on the keyboard. The only trouble is that the number changes when ANY key is pressed on the keyboard not only when keys in column one are pressed! The reason for this is that at the end of every BASIC statement a routine is called that checks to see if the BREAK key has been pressed and this is altering the setting of the PIA in between the POKE and the PEEK. The only solution is to disable the break key scan routine. How this is done involves an understanding of assembly language and so is beyond the scope of this book, being one of the topics covered in its companion volume, "The Language of the Dragon". Meanwhile, even if you cannot fully comprehend it, the following subroutine will disable the break key -

```
1000 POKE &H19B,&HE4
1010 POKE &H19C,&HCB
1020 POKE &H19D,&H04
1030 POKE &H19E,&HED
1040 POKE &H19F,&HE4
1050 POKE &H19A,&HEC
1060 RETURN
```

and the following routine will enable it -

```
2000 POKE &H19A,&H39
2010 RETURN
```

If you add 5 GOSUB 1000 to the start of the previous program (along with the lines of BASIC that make up subroutine 1000!) then the program will work as predicted in that only the keys in column 1 will alter the number returned by the PEEK. To stop the program you will have to press the reset button because the BREAK key will have no effect! You can use the BREAK disable/enable routines for other purposes but note that the BREAK key will always work during an INPUT. The direct reading of the keyboard can be useful in games because you can detect more than one key press at a time. To see this try pressing more than one key at a time while running the last example.

There is one annoying feature of the Dragon's keyboard, namely the lack of a repeat key, that it is possible to do something about with software. For many reasons a repeat facility on a keyboard makes it much more useful. When the keyboard is scanned the state of the keys in each column is stored in a different memory location, collectively known as the keyboard rollover table at &H150 to &H159. A key press will only be recognised if it changes what is already stored in the rollover table. This is the reason that the Dragon's keys do not auto-repeat. When you press a key this sets a 0 in the corresponding location in the rollover table. If you keep the key pressed nothing changes and so the fact that the key is pressed is ignored. To register another keypress you have to release the key so that the bit in the rollover table is set back to 1 and then press it again. It is possible to created an auto-repeat facility by simply regularly setting the rollover table to all 1s. First try the following program -

```
10 A$ = INKEY$
20 IF A$ = "" THEN GOTO 10
30 PRINT A$
40 GOTO 10
```

If you press a key and keep it pressed you will only see one character appear on the screen. To make another character appear you have to release and press the key again. Now add the following lines to the program -

```
  35 GOSUB 1000
1000 FOR I = 0 TO 9
1010 POKE &H150 + I,&HFF
1020 NEXT I
1030 RETURN
```

You will find that all the keys now auto-repeat once for every execution of subroutine 1000. This technique is absolutely essential if you are trying to write games in BASIC that involve using the keyboard and cursor keys to move objects on the screen. (See "The Dragon 32 Book of Games" by Mike James, S M Gee and Kay Ewbank for many examples where this technique is put to good use.)

It is possible to use the same technique to add auto-repeat to the Dragon for entering and editing programs but this requires the use of assembly language and a knowledge of interrupts and therefore cannot be fully explored in this volume. We will however return to this problem a little later in this chapter.

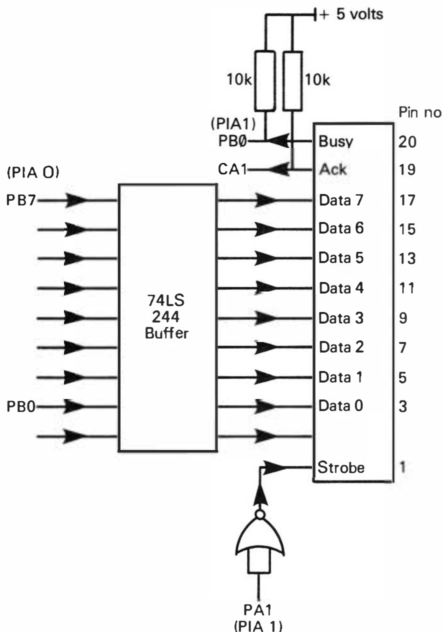
The printer interface - a user port.

The B side of PIA0 serves a dual purpose both as the keyboard column lines and as the eight output lines used by the parallel printer interface, see Figure 7.2. There is no conflict between these two uses because the keyboard is never in use when the printer is and vice versa. To be more exact, the printer output lines are active when the keyboard is being scanned but the printer ignores them because the lines that indicate that it is being used are handled by PIA 1 and are inactive. Similarly, when the printer is in use the keyboard column lines are active but the Dragon is busy printing and doesn't bother to read the keyboard inputs. Either way, the printer and keyboard sharing the same output lines does no harm and it certainly enhances the Dragon to have a printer port.

The use of the printer interface is fully catered for by Dragon BASIC with LLIST to list a program and PRINT # 2 to direct output to it. However there are a number of memory locations that affect the way the software printer drivers work that are worth knowing about, see table 7.2.

Most of these memory locations are self-explanatory. For example, if you want the Dragon to send carriage return and a line feed at the end of each line then POKE &H14A,2 will do the job.

What is more interesting about the printer interface is that it gives us the opportunity to use eight output lines for connection to other pieces of equipment. In this sense the printer interface can be used as a sort of primitive user port. Obviously the details of such a connection depend on the equipment



Pins 6,8,10,12,14,16,18 all 0 volts
Pin 2,4 + 5 volts

Fig 7.2 Printer Interface

Table 7.2

address	default	function
14A	1	number of character in end of line sequence 1 = carriage return only 2 = carriage return followed by line feed
148	FF	auto EOL when buffer full FF = no EOL when buffer full 00 = EOL when buffer full
14A -14F		EOL sequence
99		line printer comma field width
9A		last comma field width
9B		line printer field width
9C		current print position

that you plan connect to your Dragon, to but most of the ideas can be seen by using a few LEDs. If you connect eight LEDs as shown in Fig 7.3 then a high PIA line will show as a dim LED and a low PIA line will show as a bright LED. To flash LED 0 try -

```
10 TIMER=0
20 POKE &HFF02,&01
30 IF TIMER<50 THEN GOTO 30
40 POKE &HFF02,&H00
50 TIMER=0
60 IF TIMER<50 THEN GOTO 60
70 GOTO 20
```

You can have a lot of fun and learn about interfacing an PIAs with just eight LEDs!

The problem of finding some PIA lines to act as inputs for a user port is more difficult. The only lines that are free are PB0 and CA1 on PIA 1 and these are available for use from the printer port.

The joystick interface - an A to D convertor

The Dragon's joystick interface is essentially a six-bit four-channel analog to digital (A to D) convertor. Although the name 'A to D convertor' sounds impressive it is surprisingly easy to understand. A functional diagram of the A to D convertor can be seen in Fig 7.4. You can see that a major component is

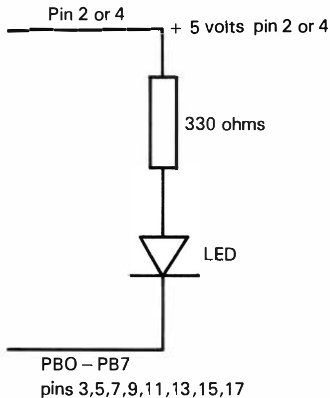


Fig 7.3 LED for printer port (eight needed)

the D to A convertor described in Chapter 5. What happens is that when a voltage is applied to the input, the Dragon produces a trial voltage using the D to A convertor to be compared with it. The output of the comparator is applied to PB7 of PIA 0 and this allows the Dragon to discover if the trial voltage is higher or lower than the applied voltage. Using this information the trial voltage is adjusted until it is as close to the applied voltage as possible. The binary number that was used to make the D to A give out this voltage is taken as the result of the conversion. This method of using a D to A convertor to measure input voltages is known as 'successive approximation' and it is a very economical way of providing an A to D.

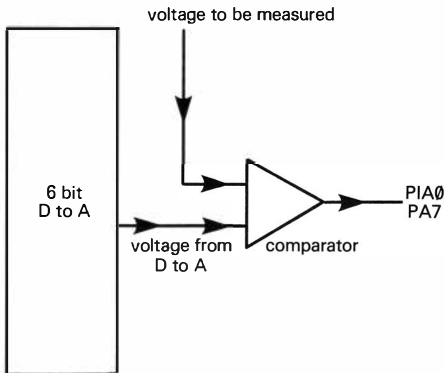


Fig 7.4 Block diagram of A to D converter

The only extra piece of hardware used by the A to D is the channel selector which selects one of the four input channels. This is in fact the same chip that selects the sound source to be sent to the TV so the methods that were used

in Chapter 5 to select the sound source could be used to select the input channel. The complete diagram of the A to D can be seen in Fig 7.5.

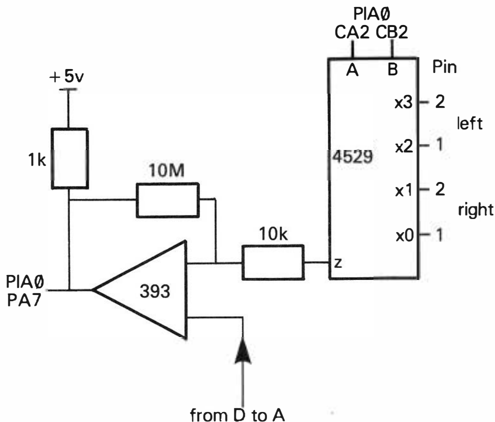


Fig 7.5 A to D Converter: The Joystick Interface

Although it would be possible to program the A to D directly there seems to be little point, as the JOYSTK function in Dragon BASIC will automatically carry out an A to D conversion for us. Where there is room for experimentation is in the connection of something other than joysticks to the joystick interface! If you have followed the discussion of how the joystick interface works you will have realised that the A to D convertor can be used to measure a voltage between 0 and 5 volts. Thus any electronic device that gives an output in this range can be connected to the joystick inputs. What makes this task even easier is that a limited 5 volt supply and an earth connection are provided on the joystick connector -

pin	function
1	channel 0
2	channel 1
3	0 volts
4	fire button
5	5 volts

The circuit of a light sensor can be seen in Fig7.6. This will return a number proportional to the light falling on the sensor and can be read using JOYSTK(0). You can connect temperature sensors, pressure sensors etc in roughly the same way. The only problem is making sure that the output of the sensor varies sufficiently between 0 and 5 volts. If this is not the case then you have no alternative but to use an amplifier. One final note of caution - DO NOT CONNECT THE JOYSTICK INPUT TO A VOLTAGE HIGHER THAN 5 OR LOWER THAN 0 VOLTS. The result of ignoring this warning is a funny burning smell coming from somewhere near the input selector! However, this should never happen if you use nothing but the 5 volt supply from the joystick interface itself.

Finally the two fire buttons, one on each joystick input, are simply connected to PA0 and PA1 inputs on PIA 0. This means that they act like additional keys on the keyboard. Moreover, pressing any key in the first two columns of the keyboard (see the previous section) has the same effect as pressing the fire buttons.

The cassette interface

There is not much that can be usefully done to alter the operation of the cassette interface from BASIC so the information included here is more for completeness than anything else. The output to the cassette recorder comes from the Dragon's sound generator which is used to produce good approximations to two audio tones - 1200Hz and 2400Hz. The input from the cassette recorder is first amplified and changed into a square wave before being applied directly to PA0 of PIA 1. The coding and decoding of the cassette signal is carried out entirely by software in the ROM. The final part of the cassette interface is the motor control circuit. This takes the form of a small transistor-driven relay connected to CA2 of PIA 1. There is no point in going into details of how to control this directly as the BASIC command MOTOR ON/OFF is very easy to use.

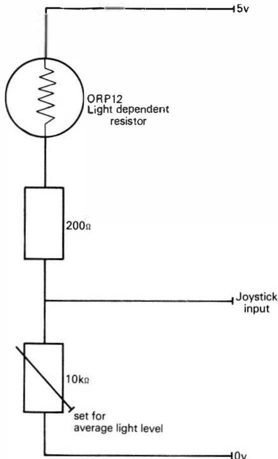


Fig 7.6 A light sensor

The BASIC timer

Dragon BASIC includes a timing function in the form of a very odd built-in variable called TIMER. If you print TIMER then it behaves like a function that returns the time in 1/50ths of a second. However, you can set the time by assigning values to it, for example, TIMER = 0, and this is more like a variable than a function. No matter how TIMER is used it is a very useful facility. The way that it works is also interesting. You might be puzzled how the Dragon manages to keep time when there has been no mention of a timer chip among its hardware. The answer is that the Dragon doesn't have a timer chip but uses the frame sync pulse from the video generator to provide a 1/50th of a second tick for a software clock.

The software clock uses the idea of an interrupt described earlier in this chapter to stop what the Dragon is doing and add one to a counter every time a frame sync pulse is detected on PIA 0's CB1 input line. You can disable interrupts from CB1 just by setting b0 in the control register to 0. First try -

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

and you will see the timer increment as usual. If you now add lines 5 and 6 -

```
5 D=PEEK(&HFF03) AND &HFE
6 POKE &HFF03,D
```

and re-run the program you will now find that the timer is stuck at zero! You will also find that the PLAY command has lost it's sense of time. If you try PLAY "C'" it will go on forever proving that PLAY uses the timer to set the length of its notes.

This is very interesting but hardly useful. However, the fact that once every fiftieth of a second the Dragon stops what it is doing and executes some timer software can be used to our advantage in some quite surprising ways. For example, in the section examining the keyboard, it was explained that an auto-repeat facility could be produced by setting every location in the rollover table to &HFF - once for each repeat. This can be done from a BASIC program in the way shown but what about while a program is being entered or edited. One solution is to add a short program onto the front of the timer software that will clear the rollover table once every 1/10th of a second. This would run whenever the timer was running and would give the keyboard a full auto-repeat facility! The only problem is that this software would have to be written in machine code. The only other information needed to write this program is that on receiving an interrupt the Dragon jumps to the machine code whose address is stored in memory location 10D and 10E.

The expansion port

The large connector on the righthand side of the Dragon that is normally used to plug in program cartridges is in fact capable of much more than this. As well as bringing out the control lines that are necessary to add extra memory in the form of ROM it also provides nearly every address, data and control line that is important inside the Dragon. Thus rather than just being a

cartridge connector it is better thought of as an expansion connector through which almost any piece of extra computer hardware can be connected to the Dragon. For example, the Dragon disk system is connected via the expansion connector.

Connecting anything to the Dragon using the expansion connector is a job for an electronics expert only, so although the pin connections and their functions are presented below, no detailed explanation is given.

Table 7.3

Pin No.	Name	Description
1	+12V	+ 12 volts
2	+12V	+ 12 volts
3	HALT	Stops 6809 and places data and address busses into 'tri-state' mode so that and external processor may take control
4	NMI	Non-Maskable Interrupt to the CPU
5	RESET	Reset and power on reset signal
6	E	E clock
7	Q	Q clock
8	CART	Interrupt input to PIA 1 CB1 Used to detect presence of Cartridge
9	+5V	+ 5 volts
10-17	D0-D7	Data bus
18	R/W	Read/Write
19-31	A0-A12	Address bus from A0 to A12
32	R2	Cartridge select signal C000 to FFEF
33-34	GND	Ground
35	SND	External sound input
36	P2	Spare select signal FF40 to FF5F
37-39	A13-A15	Address bus from A13 to A15
40	EXTMEM	Disables internal device selection

The only other point worth mentioning is that the Dragon's expansion interface is identical to the Tandy Color Computer's cartridge slot except for PIN 1 which on the Tandy machine carries -12 volts. This should be kept in mind if you intend trying to use accessories intended for the Tandy machine on the Dragon.

Software v Hardware - Using the Dragon as a VDU

One of the most interesting things about the Dragon is that it tends to use software to achieve things that other machines would use hardware for. For example, the Dragon's cassette interface, sound interface, D to A, A to D etc are all created using PIAs, the minimum of extra hardware and some clever software. In many ways this approach sums up the modern approach to electronics - if you can write it as a program then do so! As an example, consider the problem of making the Dragon into a VDU so that it can communicate with another computer or make use of Prestel. There are two apparent deficiencies in the Dragon's hardware to overcome before this is possible. First, the Dragon lacks a serial interface of any kind and most computers communicate using a serial rather than a parallel interface. Second, the Dragon lacks a text display with upper and lower case and the 32 column screen is a little restrictive. The hardware solution to these to problems is very possible but fairly expensive. The lack of a serial interface could be solved by designing an add-on to the expansion connector. (It would essentially be an 6850 ACIA chip and a few decoders and line drivers.) The lower case problem could be solved by a modification inside the Dragon which consisted of an additional character generator ROM for the video generator to use. To overcome the problem of the narrow 32 column screen would require an almost complete re-design of the Dragon or, once again, an additional circuit plugged into the expansion connector. (This would have to be a generator with its own memory etc.) Even after all this hardware was built and tested you would still have to write quite a lot of Dragon software to use it!

The alternative approach is to continue the Dragon's philosophy of using software and PIAs. The serial interface could be implemented using PIA 1 to provide one output line PA1 and one input line PB0 or CA1 from the printer port. As a serial signal is just a series of pulses (see Fig 7.7) with a regular format it can be produced by setting the output line high and low at the correct times. For input the problem is slightly more difficult but really comes down to sampling the input at regular intervals following the start bit to discover if the line is high or low. The only hardware required for this software serial interface would be a single chip to change the 0 to 5 volt output that the PIA produces to the -12 and +12 volts that is the standard for serial communication.

The lower case character problem could be solved by the use of the machine code equivalent of draw strings in a PMODE 4 to define a new

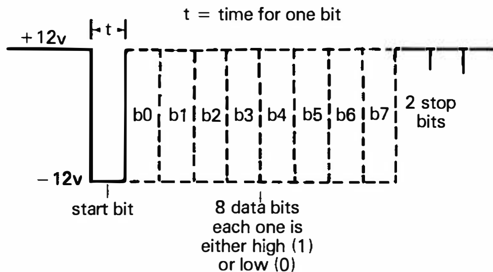


Fig 7.7 Serial data format

character set. The narrow screen could even be improved upon by using character shapes only 6 dots wide giving around 50 characters to a line. However this would reduce the quality of the display in favour of quantity.

The main point to notice is that while the first approach involved a great deal of hardware development together with a great deal of software development, the second approach is almost entirely software and as a result it is cheap to reproduce once you have finished and tested your programs! The only snag is that most of the software would have to be written in assembler to achieve the necessary speed. But as you now know enough about the Dragon to know what the programs would have to do, half the battle is already won.

Chapter Eight

Inside BASIC

In normal use there is generally no need to worry how Dragon BASIC works, it is enough that it does! However, what we have learned about the way the Dragon works makes it worth examining some of the internal organisation of BASIC. If you know how data is stored in memory then you have the opportunity of changing it directly, as in the example of user-defined characters and PUT in Chapter 6. If you know how a BASIC program is stored and processed then you have the opportunity to alter the way that BASIC works and add new commands. Finally, an understanding of the internal operation and organisation of BASIC can suggest ways of saving memory or time.

BASIC's use of memory

BASIC divides the available RAM into six regions as shown in Fig 8.1. The actual location and size of each of these regions changes according to the program being edited or run. The addresses of the divisions named in Fig 8.1 can be found by examining the appropriate pair of memory locations in low memory.

name	address pair
START	&H19,&H1A
VAR	&H1B,&H1C
ARRAY	&H1D,&H1E
END	&H1F,&H20
STACK	&H21,&H22
S.TOP	&H23,&H24
HIMEM	&H27,&H28

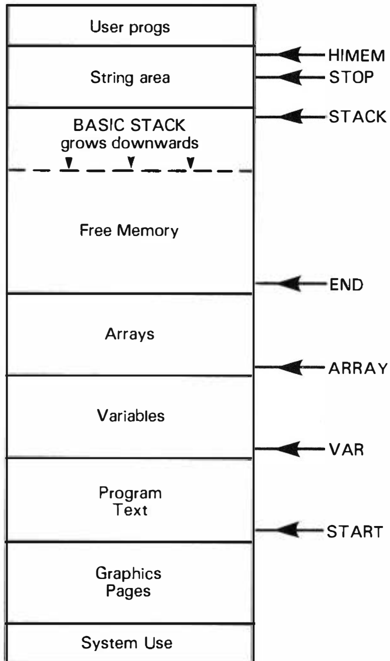


Fig 8.1 BASIC's use of memory

To find out the address stored in any of these pairs of memory locations use -

```
DEF FNN(A)=PEEK(A)*256+PEEK(A+1)
```

where A is the address of the first of the two memory locations.

The area of memory called 'program text' is where the lines of a BASIC program are stored, both as you type them in and while the program is running. START marks the beginning of this area and this moves up and down depending on the number of graphics pages reserved using PCLEAR. The next area is used to store the simple variables, real numbers and strings, that are created while the program is running. The start of this area is immediately after the end of the program text and is marked by VAR. Obviously the larger the text of a BASIC program is, the less memory there is for variable storage. Similarly ARRAY marks the start of the area of memory used to store any arrays that are created while the program is running. Notice that as the array area immediately follows the simple variable area, creating new variables after you have declared any arrays means that they all have to be moved up to accomodate the new variable. To see this, first try -

```
10 TIMER=0
20 A=0
30 B=0
40 C=0
50 D=0
60 E=0
70 PRINT TIMER
```

which will print the approximate time it takes BASIC to create and zero five new variables. The result should tell you that it takes less than a fiftieth of a second for five variables. If you add -

```
5 DIM X(4500)
```

and re-run the program you will discover that the time to create and zero five variables shoots up to around 130/50 ths of a second, or approaching three seconds! Admittedly, it is unusual to use such a large array or create very many simple variables in a program but the time it takes to move arrays around should be kept in mind if you are trying to make your programs run as fast as possible. The solution is to initialise any variables that you are going to use in your program before you dimension any arrays.

The highest memory location used by BASIC for text, variable or array storage is marked by END. The difference between END and START will tell

you how much memory your program is using for storage. To include the variables and arrays that your program creates in the result of this calculation all you have to do is run the program and then print the difference between END and START in direct mode after the program has finished. The reason that this works is that all the variables and arrays that a program created while it was running are preserved until you type NEW or add or edit a line of the program.

The highest location used by BASIC is marked by HIMEM. This can be made less than the actual physical top of memory to provide space for machine code programs and other non-BASIC uses. Just below HIMEM is the start of the string temporary storage area. When a string is defined as part of the program, e.g. A$ = "THIS IS A STRING", the characters that make up the text of the string are not copied to another part of the memory. Instead a string variable is set up in the variables area of memory which consists of a 'pointer' to the characters of the string that are stored in the program text. In this way there is no need to make a second copy of the text and so use up extra memory. However, some strings are not defined as part of the program text, they are created while the program runs. For example C$ = "*" + A$ is a string that doesn't occur in the program text. To cater for such strings the characters that correspond to its text are stored in the temporary string area. As the temporary string area is used up, the first free location is marked by S.TOP. It is possible for this area of memory to become completely full and S.TOP will try to use an area of memory set aside for the BASIC stack and an O.S. (out of storage) error will be reported. You can reserve as much memory as you need for temporary string storage using the CLEAR statement which moves the BASIC stack down in memory, (i.e. it moves STACK lower). Dragon BASIC is quite clever about the use of this string storage area. If a string is assigned a new value the old value is left in the storage area and so takes up space even though it is no longer needed. When the storage is used up the Dragon will re-organsise the temporary string area and throw away any string values that are redundant, so freeing the space that they occupied. This process is called 'garbage collection' and is the reason why the Dragon sometimes seems to pause during extensive string handling.

The final area of memory, the BASIC stack is used for the temporary storage of information such as the line number to return to following a GOSUB.

Dragon data formats

Dragon BASIC is fairly limited in the number of types of data that it can handle. Most versions of Microsoft BASIC (which is what Dragon BASIC is) can support integers, real numbers, double precision numbers and strings. However, the Dragon can only handle real numbers and strings. Not only is it limited to these two types of data, it only provides arrays with a maximum of two dimensions. For most purposes these simple data types are quite sufficient.

As already described, simple variables, real numbers and strings are stored in the region of memory beginning at VAR. Both real numbers and strings take seven memory locations to store. The format of a real number is -

m	m+1	m+2	m+3	m+4	m+5	m+6
first letter	second letter	exponent	four byte mantissa			

Notice that only the first two letters of the variable's name are stored. The format for a string variable is -

first letter	second letter	number of charts	address of text			

A string is distinguished from a real number because the second letter of the name has 128 added to its ASCII code. In other words you can find out if a variable is a number or a string by testing the value of the second letter. The address of the text is a pointer to the characters that make up the string. As already described in the last section these characters are either stored in the text of the BASIC program or in the temporary string area.

The format for an array is the same irrespective of whether or not it is a numeric or a string array. Every array is made up of two parts, a header and the data values. The header describes the type and format of the array and the data values are simply a collection of five byte values in the same format as the last five bytes of the simple variables storage format. The header format is -

first letter	second letter	array length		number of dims	0	max of first dim	

Once again, only the first two letters of the name are stored and string arrays are distinguished from numeric arrays by having 128 added to the ASCII code of the second letter. The array length is the total amount of storage allocated to the array including the header and the data values. Thus, if m is the address of the first byte of the header, m + length is the address of the first byte of the NEXT array. The number of dimensions is simply 1 or 2. If the array is one-dimensional, the next two bytes record the maximum value for the array index. If the array is two-dimensional, the next two bytes record the maximum value for first array index and there are two additional bytes used to record the array index for the second array index. Thus the header for a two-dimensional array is two bytes longer than for a one-dimensional array.

Using this information you can manipulate data stored in variables and arrays directly. In particular, you can PEEK variable locations to find out what is stored in them. It is usually not necessary to search though the variables or array area of memory for a particular variable because BASIC provides the VARPTR function that will return the address of any variable, even an array element. One use of the data formats is to write a variables and array dump program. For example the following subroutine will print out the names of all the variables that a program has used -

```
1000 V=256*PEEK(&H11B)+PEEK(&H1C)
1010 AR=256*PEEK(&H1D)+PEEK(&H1E)
1020 PRINT HEX$(V);":";
1030 PRINT CHR$(PEEK(V));
1040 IF PEEK(V+1)>127 THEN PRINT CHR$(PEEK(V+1)-128);"$";
ELSE PRINT CHR$(PEEK(V+1));" ";
1050 PRINT" ";
1060 FOR I=1 TO 5
1070 PRINT HEX$(PEEK(V+1+I));" ";
1080 NEXT I
1090 V=V+7
1100 PRINT 1110 AR=256*PEEK(&H1D)+PEEK(&H1C)
1120 IF V>=AR THEN STOP
1130 GOTO 1020
```

This program can be extended quite easily to print out the details of arrays as well as simple variables but this is left for you to work out for yourself.

The format of BASIC lines

BASIC programs are stored in memory in a compressed format that allows the Dragon's interpreter to carry out your program faster than if it was stored exactly as you had typed it. This compression takes the form of replacing keywords and functions by a single byte code called a 'token'. Apart from this each line is stored exactly as typed.

The format of a line of BASIC is -

link to next line	line number	token	rest of text	0

The first two bytes of each line contain the address of the start of the next line. This allows the BASIC interpreter to search through the memory to find any given line without having to read the whole of each line. The next two bytes contain the line number and the fifth byte contains the token that replaces the keyword. After this the rest of the text of the line is stored in as many bytes as required. Notice that although most of this text is just an ASCII representation of what you typed, any functions within the line are also replaced by tokens. The end of the line is marked by a 0 byte. The final line of a program is marked by a link value of 00.

Recovering programs

One of the many uses of knowing the format of a BASIC program and the layout of the Dragon's memory is the recovery of programs that have been accidently deleted by typing NEW. In an ideal world this sort of accident should never happen but if you ever do lose a few hours valuable work in this way then you might feel it worth trying the following procedure.

When a program is deleted by NEW all that happens is that the link bytes of the first line of the program are set to zero and the pointers that divide the memory into the various regions are reset. To recover a program what you have to do is restore the link bytes to point to the next line and set the pointers to the correct division of memory for the program. To do this you have to scan through what used to be the program storage area until you find the 0 byte marking the end of the first line. This then gives you the address of the first byte of the second line of the program which can be POKEd into the link bytes of the first line of the program. Scanning though memory can take some time

if the first line of your program was at all long. Start at the memory location given by PEEK(&H19)*256 + PEEK(&H1A) and enter all commands in direct mode i.e. do not use line numbers and RUN a new program because this will overwrite the program you are trying to recover. Once you find a memory location containing zero and have POKEd the address into the two link bytes of the first line you will find that you can list the program. However, do not try to run it nor add any lines because, although you can list it, the VAR pointer is still set to the wrong place. Your next task is to locate the address of the end of the program. To do this you need a short program typed in direct mode, but first you will have to raise the area where variables are stored away from the program area. First set the the VAR pointer to something very large and well outside the old program area and then type the following single line program in direct mode.

```
FOR I= start TO &H7FFF:PRINT I:I = PEEK(I)*256 + PEEK(I + 1)
-1:NEXT I
```

where 'start' is the start address of the text area. The one line program will print out the value stored in the link byte of each line. When you find a link byte that is zero the address of the last line of the program will be the number printed just before the zero. Use this address to find the end of the line and then adjust the values of VAR, ARRAY and END to point to just beyond the end of this line.

Although certainly worthwhile if ever you do lose a long program by mistake, this procedure is long and difficult. There are many things that can go wrong with it and my advice is to save programs often and hope that you never have to use it!

The TAB function

If you examine the BASIC ROM in any detail you will quickly find an area of memory that stores the BASIC keywords and functions. If you look carefully you will notice that there is a keyword that isn't mentioned in the Dragon manual - TAB. If you try ,

```
10 FOR I=0 TO 15
20 PRINT TAB(I);"X"
30 NEXT I
```

you will find that the Dragon does indeed have a TAB function that moves the cursor from its current printing position to column I.

Adding commands to BASIC

It is possible to add new commands to BASIC and so to a certain extent produce customised versions of Dragon BASIC. However, this is a procedure that can only be undertaken in assembler and so this next section will not be of use to the BASIC only programmer.

The way that new commands can be added to BASIC involves intercepting the 'get next character' routine used by the BASIC interpreter to process a program while it is running. This routine is a machine code program that is loaded into low memory when the Dragon is first switched on. It is a self-modifying program so what it does can be a little tricky to work out.

Address		assembly language
9F	GETCHAR	INC A7
A1		BNE SKIP
A3		INC A6
A5	SKIP	LDA 0000
A8		JMP BB26

The first part of the program adds one to the address stored in memory locations A7 and A6. If you look carefully you will see that these two locations are in fact within the program and form the address field following the LDA instruction. The effect of this is to load the A register from one location further on each time the routine is used. Thus the BASIC interpreter can set location A7 and A6 to the first character of the BASIC program and use the routine whenever it needs the next character.

To add a new command all you have to do is change the JMP BB26 instruction to point to the assembly language routine that implements your new command. The first part of this routine would check the current character in the A register to see if it could be the first letter of the new command. If it is then the next character would be checked for a match and if they did the assembly language routine would carry out the command. If they didn't match then the character would be passed on to the interpreter by executing a JMP BB26 instruction.

This method isn't the only way of adding commands, you could intercept the error handler for example, but it is one of the easiest. Whatever method is used there is no way that assembly language can be avoided.

Some useful memory locations

There are a number of memory locations that BASIC uses either to store temporary data or the addresses of routines that it uses while running a program. Some useful data locations are given in table 8.1 -

Table 8.1

address (hex)	function
2D,2E	Address of current BASIC line
2F,30	Address of start of BASIC text used for warm restart following reset
31,32	DATA line number
33,34	DATA memory pointer
68,69	Current line number
6F	Current device 0 = VDU,-1 = cassette,-2 = printer
71	Restart flag if &H55 then warm start
78	Cassette status 0 = closed,1 = input,2 = output
87	Last key pressed (may be cleared by break check)
88,89	next location for screen output
9D,9E	Address used by EXEC
B2	Current foreground colour code
B3	Current background colour code
B4	Colour in use
B6	Graphics mode
B7,B8	Address of top of graphics screen
B9	Number of bytes in a graphics row
BA,BB	Base address of current graphics screen
BD,BE	Current x co-or dinate
BF,C0	Current y co-or dinate
112-113	Timer

Many of BASIC's useful routines call dummy subroutines in lowmemory. These subroutines consist of RTS instructions so that normally they have no effect. However, as each has three bytes allocated to it you can change the RTS instruction to a JMP to a user routine and so modify the the way that BASIC works. Some useful examples of these dummy subroutines are given in table 8.2 -

Table 8.2

address (hex)	called from
167	Input a character
16A	Output a character
18E	Error handler
194	RUN
19A	Read next state ment, (after returning a check for break is carried out, so increasing the return address by 4 will disable break)

On from BASIC

To release the real power of the Dragon's hardware there is nothing better than assembly language. However, for ease of use there is nothing better than BASIC. Considered overall, Dragon BASIC is not a bad implementation of BASIC and it is probably better to proceed by writing as much of a program in BASIC as possible. To tackle the tasks that need assembler the USR or EXEC commands (introduced in the next chapter) can be used to good effect to call assembler routines. There is a lot to be said for the point of view that it is better to spend time adding new commands to Dragon BASIC rather than just writing one-off routines. If you write an assembly language routine only other assembly language programmers can use it, but if you add it to BASIC as a new command, anyone can use it.

Chapter Nine

Introduction to 6809 Assembler and Machine Code

The Dragon is a remarkable machine and especially so when you take into account its competitive price. However there are applications that Dragon BASIC just isn't up to. This isn't a problem that is specific to the Dragon, it is inherent in the way that BASIC is implemented on small computers. The main complaint against Dragon BASIC is that it is too slow. There are many occasions when this lack of speed is tolerable. If a number crunching program is too slow then you can always go and have a cup of coffee while you wait! The problem only becomes serious when the waiting time grows to days or when you need an almost instantaneous response. For example, it is possible but very difficult to write a space invader style program in BASIC but, even if you try very hard, you will still find it difficult to make anything move quickly and smoothly on the screen using BASIC. There is also a second problem with BASIC that has often become apparent in earlier chapters. If you want to alter the way BASIC works or add to it then you cannot use BASIC. No matter what you are interested in, sooner or later you will run up against the limitations of BASIC.

You may think that these limitations are something to do with the fact that the Dragon is just a home computer. You would be wrong. The reason is that *ALL* micro computers, perhaps *ALL* computers whatever their size, eventually prove too slow for some task that they are given! Before deciding that the Dragon is too slow for the range of applications you want it to undertake it is only fair to give it a chance to show what it can really do. The Dragon has hidden power and is capable of running your programs much faster than you would expect. The cost of this speed increase is, however, quite high in that you have to abandon BASIC and learn 6809 machine code. Despite any arguments to convince you otherwise, machine code *IS* more

difficult than BASIC (why else would anyone use BASIC!) and it takes a certain amount of dedication to come to terms with it. Now this may sound like advice to avoid machine code like the plague but in fact the rest of this chapter should encourage you to learn machine code for yourself. The point is that machine code isn't something that you can pick up in an afternoon but it is a rewarding thing to learn.

Slow BASIC.

BASIC is one of the many 'high level' languages that you can use to program a computer. Any given computer can often be programmed in a range of such languages e.g. BASIC, FORTRAN, ALGOL etc. For example, the you can buy a variety of program packages for the Dragon that give you the Forth programming language. The way that one computer can run so many different languages is that each one is translated to a more fundamental language before the program is run. This more fundamental language is usually called machine code and it is the only language that a computer can obey directly. Each different machine has its own machine code language and each machine translates the high level languages available for it into its own personal code. This means that you cannot learn machine code in general but only a specific computer's machine code. The Dragon has a 6809 microprocessor inside it so the machine code that it uses is 6809 machine code. This is a very good choice for a first machine code to learn because the 6809 is very well designed and logical but you should be aware that many other well-known computers such as the Sinclair Spectrum, the BBC Micro, APPLE and PET do not use a 6809. Indeed the Dragon is one of the first home computers to use it.

As already explained, your Dragon has to convert your BASIC statements to machine code before they can be carried out. In fact the process is rather more subtle than a direct translation to machine code. There are machines that translate a whole BASIC program to machine code before carrying it out by the use of a program called a 'compiler'. However these machines are in general more difficult to use than the Dragon which uses a different technique. What happens inside the Dragon when you run a program is that each line of BASIC is examined at the moment that it is to be carried out. The keyword is then used to 'look up' what is to be done in a list of actions. For example, if the line of BASIC was GOTO 10 then the Dragon determines that the keyword is GOTO and uses this to look up what to do in a table. The entry in the table for GOTO would contain the machine code equivalent of:

"obtain the number following the GOTO, find the line of BASIC with the same line number and make this the next instruction to be obeyed".

This method of running a BASIC program is known as 'interpreting' and the program that carries it out is called an 'interpreter'. Thus the Dragon interprets every line of BASIC that you write and this is why BASIC is so slow. Before the action that your BASIC command specifies can happen, the Dragon has to spend a lot of time working out what your line of BASIC actually means! By contrast, a machine code program is executed immediately without any interpreter and can therefore often run over ten times faster.

The characteristics of machine code

If machine code is so much faster than BASIC why don't we use it more often. The answer to this question has already been briefly mentioned in the introduction to this chapter - it is more difficult to program in machine code than in BASIC. The reason why machine code is more difficult than BASIC is that it is a much simpler language! In BASIC you might write something like A = B + C/2-Z and rightly expect the answer to be stored in A, but in machine code the only arithmetic operations that you can use are addition, subtraction and multiplication and these can only be carried out one at a time and on numbers small enough to be stored in one, or at most two, memory locations! To do 'difficult' things like division you have to write subroutines that will split them down into simpler operations. For example, to divide one number by another you have to resort to a form of repeated subtraction.

It is not within the scope of this book to teach you machine code but the rest of this chapter will attempt to give you the 'flavour' of machine code programming and an introduction to some of the fundamental ideas involved. The best way to achieve this is via a couple of simple examples. First it is necessary to examine some of the details of the 6809 microprocessor used in the Dragon.

The 6809

The 6809 microprocessor is one of the fastest and most powerful available today but it is still only capable of very simple operations. It can access any of the memory locations that we have discussed in earlier chapters but it has few

instructions which alter them directly. Any operations have to be carried out in special memory locations inside the microprocessor itself. These special memory locations are called 'registers' and because there are so few of them they are referred to by names rather than addresses. The 6809 has a number of registers but for the sake of simplicity we will examine and use only three -

the A register
the B register
and the X register

The A and B registers are almost identical in the range of things that you can use them for. They are involved in nearly all the arithmetic operations that the 6809 can perform. They are exactly like an ordinary memory location in that they can only hold numbers between 0 and 255. The X register is capable of fewer operations than the A and B registers but it is twice as large as either of them. It can hold the equivalent of two ordinary memory locations and so the range of numbers that it can handle is considerably increased. In fact the range is large enough to hold the address of any memory location in the Dragon; and this is the main use of the X register - as an address register. Many 6809 operations employ the number stored in the X register as the address of the memory location that will be used.

All this talk of registers and what they are used for is easier to understand after one or two examples of 6809 instructions.

The LDA A and ADD A instructions

Machine code is recognised by the 6809 in terms of numbers. That is, a machine code program is nothing more than a long list of numbers stored in the computer's memory. (After all what else but a number can you store in a computer's memory!) The trouble is that humans are very poor at reading through and understanding long lists of numbers. So to make machine code easier to use we use convenient symbols instead of the numbers that the computer recognises to represent operations. For example, if you want to load the A register from memory location &H12 you would use a version of the LDA A instruction -

LDA A &H12

which you can read as "Load the A register from memory location &H12". However, a 6809 cannot read this type of code so the instruction has to be coded into numbers. If you look up LDA A in a list of 6809 instructions you will find that its code is &HB6. This is referred to as the 'operation code' or 'op code' for LDA A from a memory location. The complete coded instruction also requires the address of the memory location to be loaded into A to be written after the op code. So the complete coded instruction is &HB6,&H00,&H12. If you're wondering why there is an additional zero byte code in front of the instruction the reason is that two memory locations are used to store the address. In other words the address is &H0012 which is stored as &H00 in one location and &H12 in the second. Thus a single, very simple, 6809 instruction takes three memory locations to store. Other 6809 instructions may take less memory, others take more.

As another example, of a 6809 machine code instruction consider the

ADD A #n

instruction where n is a number in the range 0 to 255. This simply adds n to the contents of the A register. That is, if A was already loaded with 3 then after ADD A #5 it would contain 8. The # is used to distinguish between a number and an address so ADD A 5 would mean add the contents of memory location 5 to the A register. This instruction has an op code of &H8B and the number to be added n is stored in the next memory location. Notice that as n is restricted to the range 0 to 255 only one memory location is required. Thus the complete coding for ADD A #5 is &H8B,&H05 and this only uses two memory locations.

The usual way of writing a machine code program is to use the symbols such as LDA A and ADD A, to write the entire program and then go through and convert it to a list of numbers. The conversion of the symbols to numbers is such a routine job that you can get a program to do it for you. Such a program is known as an 'assembler' and it certainly makes machine code programming easier. Unfortunately the Dragon doesn't have an assembler as a standard extra although there are some available for it on the market.

A short example program

There is no suggestion that after the last two sections you'll know enough about machine code to be able to write a program but you should be able to understand the outline of one. There is a machine code subroutine in the

BASIC ROM that starts at memory location &H800C that will print the character whose ASCII code is stored in the A register at the current position on the screen. We can use this knowledge to write a very simple program that will fill the screen with any character of our choice.

```
START LDA A #&H41
JSR &H800C
JMP START
```

The first instruction in this program loads the A register with &H41, the ASCII code for 'A'. The second instruction is like a machine code equivalent of GOSUB in that it transfers control to a machine code subroutine starting at &H800C. (JSR stands for Jump to SubRoutine.) As we already know, the subroutine at &H800C prints the character whose code is stored in the A register. You should be able to see that the result of this JSR is a 'A' printed on the screen. The final instruction is a machine code equivalent of GOTO in that it transfers control to the 'START' of the program. (JMP is short for JuMP so you can read the last instruction as 'jump to START'.) The workings of this program are not difficult to understand - it will simply print 'A' on the screen until it is full and then it will continue with the screen scrolling after each line of As until the reset button is pressed. Normally we have to worry about stopping machine code subroutines after they have finished what they are doing but in this case, for simplicity, the reset button will do the job.

We now have a fully specified machine code program. The only things that remain to be done are coding and testing. Coding, if you recall, is simply changing the symbols that humans use to write machine code into the number codes that computers understand. The first two lines of the program can be coded easily -

Instruction	Code
START LDA A #&H41	&H86,&H41
JSR &H800C	&HBD,&H80, &H0C

The code for LDA A with a number is &H86 and the number to be loaded follows the op code i.e. &H41. The op code for JSR is &HBD and the address of the subroutine follows in the next two memory locations. The last instruction presents a problem however. The op code for JMP is &H7E and the address that it transfers control to is stored in the next two memory locations - the trouble is we don't know the address of the start of the program!

To know the address of the start of the program we have to decide where it is going to be stored in memory. There are a number of places that machine code can be stored in the Dragon and each has advantages and disadvantages. The simplest solution is to reserve some memory using the CLEAR command-

```
CLEAR s,h
```

will reserve s bytes for string storage and reserve memory from address h + 1 up for machine code. (That is, the highest address that BASIC will use is h.) Once we have reserved some memory we still have the problem of getting the machine code into it, after all unlike BASIC machine code cannot be entered directly into memory from the keyboard. One of the simplest methods of transferring a list of numbers to any memory locations is to use a DATA statement to hold the numbers and use a POKE instruction to transfer them one by one into memory. For example if the list of numbers is stored in DATA statements before line 50, the following lines of BASIC -

```
50 FOR I= S TO S+N-1
60 READ D
70 POKE I,D
80 NEXT I
```

will transfer N numbers into memory starting from the location whose address is stored in S. So if we reserve memory above &H7000 for machine code programs (this is around 1000 memory location which is more than enough for the simple demonstration programs in this chapter!) and store the program starting at &H7001, then the address corresponding to START is also &H7001. This means that the full form of the JMP instruction is

```
JMP &H7001
```

which can be coded as

```
&H7E,&H70,&H01
```

We can now write the final list of numbers that corresponds to the machine code of the program in a DATA statement and use the READ and POKE method of storing the list in memory. That is,

```
10 DATA &H86,&H41,&HBD,&H80,&H0C,&H7E,&H70,&H01
20 CLEAR 100,&H7000
30 FOR I= &H7001 TO &H7001+8
40 READ D
50 POKE I,D
60 NEXT I
```

This program will take each of the numbers in the DATA statement and store them in memory but that is all. We still have to find a way of making the Dragon obey this list of instructions. There are two different ways of doing this the USR function and the EXEC command. For our purposes the EXEC command is simpler and sufficient

```
EXEC address
```

will transfer control to the machine code program starting at 'address'. As we already know the address of the start of the program all we have to do is add

```
70 EXEC &H7001
```

to the program.

When you run this final version of the program you should see the screen fill very rapidly with letter As. To compare the speed of this machine code with BASIC add the following lines and then enter GOTO 80

```
80 PRINT "A";
90 GOTO 80
```

Although this example, has been very simple, consisting of only three machine operations, it has taken a long time to produce and a long time to explain! However, you should be able to judge the advantages of machine code from the speed difference between this example, and the BASIC equivalent.

A second example, - reversing the screen

In this example, we will write a machine code subroutine that will change all of the characters displayed on the screen to their inverse form (i.e. green will be changed to black and black to green). Although this is a useful subroutine, it could be used to 'flash' the screen during a game for example, its primary purpose is to show how a slightly larger machine code program is written. However, as the program is so much longer there just isn't the space to go into as much detail as with the first example. All that can be done is to explain the method used and how each instruction works. The details of coding are given but not explained.

The method underlying this routine is to change bit 6 of every text screen memory location to a 0, if it is initially a 1, and to a 1 if it is initially a 0. If you go back to Chapter 3 you will find that bit 6 controls which way round, black on

green or green on black, a character will be displayed and so c the two numbers were different. For example, the exclusive or of 1011 and 1110 is 0101 i.e. there is a 1 in the answer only where the two numbers are different. To reverse bit 6 in a number all you have to do is to exclusive or it with &H40. The reason that this works is that bit 6 in &H40 is 1 and all the other bits are 0 and so if bit 6 in the first number is 1, bit six in the answer will be 0 and vice versa, while all the other bits in the first number are left unaltered. To see this, try working out the exclusive or of 01011101 with &H40 or 01000000 (the answer is 00011101).

The rest of the method is straightforward. We have to load the X register with the address of the start of the screen area and then use this to load each screen memory location into A, exclusive or it with &H40 and store it back into the same location. The only thing that we have to be careful about is to avoid 'running off' the end of the screen. The best way to achieve this is to test the value of X each time one is added to it to move it on to the next screen location. The only other detail to remember is that all machine code programs should return to BASIC by using a RTS instruction. (RTS stands for ReTurn from Subroutine and is similar in function to the BASIC RETURN.)

The complete program along with some explanatory comments is -

	LDX #&H400	Load the X register with the address of the start of the screen
LOOP	LDA A 0,X	Load A with the memory contents of the memory location whose address is in X
	EOR A #&H40	Exclusive OR of A with 40
	STA A 0,X	Store the A register back in the same screen location
	LEAX 1,X	Add one to the X register
	CMP X #&H600	Compare the contents of X with &H600
	BEQ +3	Skip the next instruction if X is equal to &H600
	JMP LOOP	Jump back to reverse next location
	RTS	Return to BASIC

To try this program out it is necessary to convert it into machine code and include it in a DATA statement as in the last example,. If you do this you the following program results -

```
    10 CLEAR 100,&H7000
    20 DATA &H8E,&H04,&H00,&HA6,&H84,&H88,&H40,&HA7,
    &H84,&H30,&H30,&H01,&H8C,&H06,&H00,&H27,&H03,&H7E,
    &H70,&H04,&H39
    30 FOR I= &H7001 TO &H7001 +19
    40 READ D
    50 POKE I,D
60 NEXT I
70 EXEC &H7001
80 GOTO 70
```

Although this is a long example, you should be able to understand most of it if you study it carefully. If you compare the speed of this machine code routine with the time it takes BASIC to print a screen full of characters you will see the reason why machine code is worth while.

Next steps

If you have managed to understand some of this chapter and have entered the examples and seen how much faster they are than BASIC then machine code will be the next area of computing that you will want to study. When you are ready to tackle this topic, look out for the companion volume to this book, "The Language of the Dragon : 6809 Assembler" in which I continue to explore all the aspects introduced in this chapter with many practical examples that complete some of the unfinished business of this book.

Whether or not you decide to pursue machine code programming, the other techniques explained in this book should enable you to get a good deal more out of BASIC on your Dragon than you might previously have thought possible. There is usually a BASIC solution to every programming problem and one lesson to be learned from this book is that it is worth searching for it. Understanding the anatomy of your Dragon should help you in this quest.

Appendix I

6847, 6821 and 6883 Pin Connections

The Video Chip

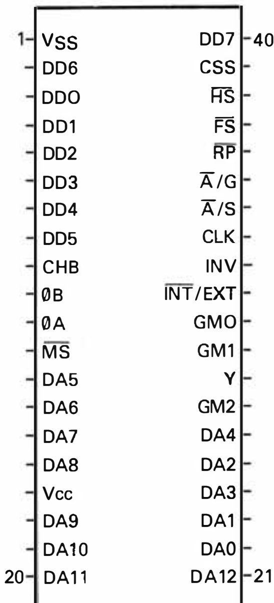


Fig A.1 6847 Video Chip

The PIA Chip

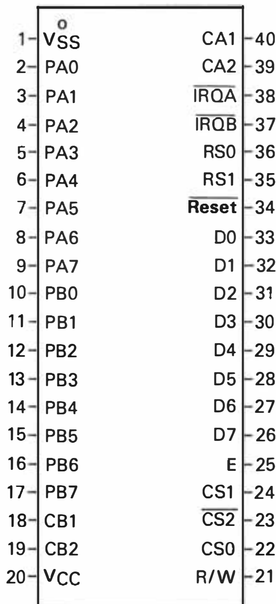


Fig A.2 6821 PIA

The SAM Chip

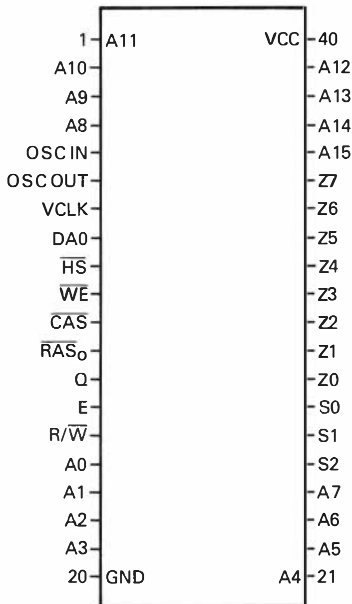


Fig A.3 6883 SAM chip

Appendix I (continued)

The complete list of SAM address pairs is

Address pair SET/CLEAR	Name	Function
FFDF/FFDE	TY	Map Type
FFDD/FFDC	M1	Memory size
FFDB/FFDA	M0	
FFD9/FFD8	R1	CPU rate
FFD7/FFD6	R2	
FFD5/FFD4	P1	Page #1
FFD3/FFD2	F6	
FFD1/FFD0	F5	
FFCF/FFCE	F4	
FFCD/FFCC	F3	Display offset
FFCB/FFCA	F2	
FFC9/FFC8	F1	
FFC7/FFC6	F0	
FFC5/FFC4	V2	
FFC3/FFC2	V1	Display mode control
FFC1/FFC0	V0	

The function of the SAM control addresses V0 to V2 and F0 to F6 have already been described in detail in Chapters 3 and 4. The rest are best left alone because they are intimately connected with the Dragon's hardware configuration. For example, the two control addresses M1 and M2 specify the type and amount of memory used and, as a standard Dragon always has 32K of dynamic RAM, there is no point in changing the default setting. However, the control addresses refered to as P1 will be of some interest if you feel like

adding some more memory to your Dragon. By setting or clearing P1 you can access one of two alternative 32K banks of memory controlled by the SAM. The exact details of this are involved and you will certainly need a full data sheet for the SAM chip before you could tackle such a project, but it is possible.

The only other SAM control addresses worth knowing about are R0 and R1. These govern the speed at which the 6809 CPU operates. While it is true that all Dragons come equipped with a 'single speed' 6809 which is only guaranteed to work at 1 MHz (or one memory access per microsecond) they will often work at a higher speed. R0 and R1 control the 6809's speed in the following way -

R0	R1	
0	0	.9MHz
0	1	.9MHz for addresses 0000-7FFF and FF00-FF1F 1.8MHz for all other addresses
1	0	1.8MHz
1	1	1.8MHz

You are unlikely to have a Dragon in which all the components work at at double speed, for one thing the SAM will only handle the dynamic RAM at .9MHz! However the mode with R0 set and R1 clear will double the speed of the 6809 when it is accessing ROM or I/O and run at the normal speed when accessing RAM. Not even this mode is guaranteed to work on all Dragons because it is possible that the 6809 itself will refuse to work at this higher speed! If you want to find out whether or not your Dragon will go faster simply POKE/ &HFFD7,0. If your machine goes silent than it won't and you will have to press reset to return things to normal. If you find that your Dragon does carry on running after this POKE then you can use it at a higher speed but remember to change back to the lower speed for saving and loading tapes, playing music or anything that involves a time component that is set by the speed that the 6809 is running at!

Appendix II

The Graphics Modes

SAM			PIA 1	resolution	colours	memory	graphics
v2	v1	v0	7 6 5 4 3 2 1 0	rows x cols			mode
0	0	0	0XX0CUUU	16 x 32	2	512	ALPHANUM
0	0	0	0XX0CUUU	16 x 32	2	512	ALPHA INV
0	0	0	0XX0XUUU	32 x 64	9	512	SEMI-G 4
0	0	0	0XX1CUUU	48 x 64	2	512	SEMI-G 6
0	1	0	0XX0XUUU	64 x 64	9	204	8SEMI-G 8
1	0	0	0XX0XUUU	96 x 64	9	307	2SEMI-G 12
1	1	0	0XX0XUUU	192 x 64	9	6144	SEMI-G 24
0	0	1	1000CUUU	64 x 64	4	1024	1F
0	0	1	1001CUUU	64 x 128	2	1024	1T
0	1	0	1010CUUU	64 x 128	4	2048	2F
0	1	1	1011CUUU	96 x 128	2	1536	PMODE 0
1	0	0	1100CUUU	96 x 128	4	3072	PMODE 1
1	0	1	1101CUUU	192 x 128	2	3072	PMODE 2
1	1	0	1110CUUU	192 x 128	4	6144	PMODE 3
1	1	0	1111CUUU	192 x 128	2	6144	PMODE 4

X means don't care,U means do not change and C selects between one of two colour sets. For details of memory maps etc see Chapters 3 and 4.

Index

A

Adding (New Commands) 120
Address Bus 8
ALGOL 124
Alphanumeric Modes 21
AND 32,91
ARRAY 114
ASCII 25
A/S Pin 28
A to D Convertor 101
AUDIO 61

B

Background Preservation 69
BASIC 1
BASIC Interpreter 118
BASIC Lines 118
BASIC's Memory 112
BASIC TIMER 106
Binary 93
Block Diagram 15
BREAK Key 97

C

CA1 87
CA2 87
Cassette Interface 105
CB1 87
CB2 87
CHR$ 33
CIRCLE DRAW 64, 73
CMOS Buffer 54
COLOR f, b 44
Control Bus 8
Control Pins 23
CPU 6
CSAVE 57

D

Data Bus 8
Data Direction Register 87
Data Formats 116
Decimal 16
DIM 78
Display Modes 21
DRAW 70
D to A Converter 50
Dummy Variable 61

E

END 114
Expansion Port 107

F

Fire Buttons 105
FNN() 52

FORTRAN 124
Frame Pulse 9
Frame Sync 9

G

GET 64,76
Graphic Modes 21,45,135

H

Hex 16,93
High Resolution Graphics 40
HIMEM 115
Histograms 31

I

Interrupts 89
INT/EXT Pin 28
INV Pin 25
Inverted Text 24

J

Joystick 83
Joystick Interface 101

K

Kaleidoscope Patterns 31
Keyboard 95

L

LDAA 126
Lightpens 83
Light Sensor 84
LINE 64,65,74
Line Pulse 9
Line Sync 9
Low Resolution Graphics 28
Lower Case Digits 27

M

Machine Code 123,125
MC6847 9
Memory Locations 121
Memory Mapping 9,16
Modulator 14
MOTOR 61

N

NEW 118
NOT 91

O

OR 91

P

Paged Graphics 64
PAINT 64,73
PCOPY 69
PCLEAR 44,64
PCLS 44,64
PEEK 24
PIA 12,86,90
PLAY 51
PMODES 41,44,49,64
POINT 31
POKE 24
PRESET 64
PRINT 27
Printer Interface 99
Program Text 114
PSET 64,74
Pulsating Circle 67
PUT 64,76

R

RAM 7
RAM, Dynamic Chips 7
RAM, Static Chips 7

RECORD 57
RESET 30
Recovering (programs) 118
Reversing (Screen) 130
ROM 7,8

S

SAM 8,9,23
SAM Address Pairs Appendix I
Scaling Procedure 84
SCREEN 44,64
Semi-Graphic Modes 21,34,39
SET 30
SOUND 50
Sound Generator 53
Source 57
Square Wave 56
START 114
Start Address 47
String Storage 115
Synchronisation 62

T

TAB 119
Tempo 52
Text 24
TV Display 8

U

User Ports 14

V

VAR 116
VARPTR 81, 117
VDU 109
Video Generator 9
Video RAM 9
Voltage 53
Voltage Transition 89
Volume 52

A message from the publisher

Sigma Technical Press is a rapidly expanding British publisher. We work closely in conjunction with John Wiley & Sons Ltd. who provide excellent marketing and distribution facilities.

Would you like to join the winning team that published this and the other highly successful books listed on the back cover? Specifically, **could you write a book that would be of interest to the new, mass computer market?**

Our most successful books are linked to particular computers, and we intend to pursue this policy. We see an immense market for books relating to such machines as:

The BBC Computer
PET
Apple
Tandy
Sinclair
Osborne
Atari
IBM
Sirius. . .and many others

If you think you can write a book around one of these or any other popular computer — or on more general themes — we would like to hear from you.

Please write to: Graham Beech,
Sigma Technical Press,
5 Alton Road,
Wilmslow,
Cheshire, SK9 5DY,
United Kingdom.

Or, telephone 0625-531035.

About This Book:-

Here is a complete guide to programming the Dragon, now that you have progressed beyond the manual. It begins with a brief review of what you ought to know already and then takes you through:

Dragon hardware: its anatomy - chips and circuitry - as it affects the programmer.
Low - Res Graphics and Text.
Hi - Res Graphics
Sound generation - from music to talking programs.
Advanced graphics - including animation.
Interfacing - joysticks, and the various input/output parts.
Inside BASIC - includes entry points and system variables.

The book ends with an indication of how machine code can help the programmer, which links into Mike James's second book for Sigma - "Language of the Dragon".

Other Books of Interest:-

Language of the Dragon, by M. James.
Hot Programs to feed your Dragon, by G.P.S. Robinson and M.A. Smith
The Complete FORTH, by A.F.T. Winfield
The Sinclair Spectrum in Focus, by M. Harrison
Practical Pascal for Microcomputers, by R. Graham
Practical Programs for the BBC Computer, by D. Johnson-Davies.

About the Author:-

Mike James is a prolific author of books and articles on computing and microcomputers. He is an experienced lecturer and now runs Infomax, a computer consultancy.

Published by
Sigma Technical Press
5 Alton Road
Wilmslow
Cheshire
SK9 5DY
U.K.

ISBN: 0 905104 35 8

Anatomy of the DRAGON

Mike James

Σ