Secrets of the COMMODORE 64

P. CORNES & A. CROSS

SECRETS OF THE COMMODORE 64

SECRETS
OF THE
COMMODORE 64

by
P. CORNES & A. CROSS

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

PLEASE NOTE

Although every care has been taken with the production of this book to ensure that any projects, designs, modifications and/or programs etc. contained herein, operate in a correct and safe manner and also that any components specified are normally available in Great Britain, the Publishers do not accept responsibility in any way for the failure, including fault in design, of any project, design, modification or program to work correctly or to cause damage to any other equipment that it may be connected to or used in conjunction with, or in respect of any other damage or injury that may be so caused, nor do the Publishers accept responsibility in any way for the failure to obtain specified components.

Notice is also given that if equipment that is still under warranty is modified in any way or used or connected with home-built equipment then that warranty may be void.

All the programs in this book have been written and tested by the authors using a model of the COMMODORE 64 that was available at the time of writing in Great Britain.

Some details of graphic modes may differ for machines for overseas markets.

First Published – April 1984

British Library Cataloguing in Publication Data
Cornes, P.
Secrets of the Commodore 64.
1. Commodore 64 (Computer)
I. Title II. Cross, A.
001.64'04 QA76.8.C64

ISBN 0 85934 110 0

Printed and bound in Great Britain by Cox & Wyman Ltd, Reading

PREFACE

This book is a beginner's guide to the Commodore 64. In it, we will be giving you masses of useful information and programming tips as well as describing how to get the best from the powerful sound and graphics facilities. The only assumption we have made is that you already have a copy of the Commodore 64 Users Manual (which comes free with the computer anyway!).

The book gets under way in Chapter 1 with a brief look at the way the memory is organised and how much of it is available to you from BASIC. Chapter 2 is all about random numbers and ways of generating them. This is important because random numbers have endless roles to play in both entertainment and serious programming applications. Chapter 3 is another general chapter, this time on two very useful and often misunderstood BASIC facilities – PEEK and POKE. Our look at the Commodore 64s' special facilities begins in Chapters 4 and 5 with a description of low resolution graphics, colour and simple animation. Chapter 6 takes us into the more advanced world of animation and looks at Sprites. Chapter 7 explains the mysteries of high resolution graphics, including plotting points and drawing lines and circles. Chapter 8 describes the use of the fabulous sound facilities, including using them for both music and sound effects. And to finish we have two very useful chapters dealing with machine code. Chapter 9 is concerned with machine code programs in general on the Commodore 64, and the final chapter presents a machine code program which extends the BASIC on the machine by adding nine additional statement keywords which can be used in any BASIC program.

If you don't fancy typing in the listing we have given, you will be pleased to know that it is available on a cassette tape*, at very reasonable cost, from GENERAL SOFTWARE,

---

*PLEASE NOTE: The publishers of this book are in no way responsible for the manufacture or supply of this tape and all enquiries must be sent directly to GENERAL SOFTWARE.

PO Box 15, STONE, STAFFORDSHIRE. The documentation supplied with the tape also explains how you can use the "core" of our software to add your own keywords to BASIC.

Phil Cornes and Tony Cross

ACKNOWLEDGEMENTS

We would very much like to thank our respective wives Ruth and Chris for their patience and assistance during the writing of this book.

CONTENTS

Chapter 1

IN THE BEGINNING

The Commodore 64 is one of a new breed of home computer. It offers many powerful new facilities, including high and low resolution graphics in up to 16 colours, Sprite graphics, an amazingly powerful facility for animation and action, and last but by no means least, an incredible 3 channel sound synthesiser.

All these facilities are available by using a few simple BASIC statements. Getting the best from these facilities however, requires a detailed knowledge of the way in which each of them works. And that is where this book comes in. In it we will show you how to access and use all the above facilities, and many more, by giving you the detailed information you need including many programming hints, tips and tricks.

Managing Memory

Let's start by sorting out just how much memory is available to us from BASIC.

As its' name suggests the Commodore 64 comes already fitted with a full 64K of random access memory (RAM). We say a full 64K because this is the maximum amount of memory which the microprocessor within the machine can directly access. Like any other computer, the Commodore 64 also needs to be able to access some read only memory (ROM). The ROM contains programs which must be kept permanently – like the operating system, which tells the computer how to do things like scan the keyboard and drive the cassette tape etc. and BASIC which is the resident high level language. In all there is 20K of ROM required on the Commodore 64 and this has to fit into the 64K address space of the microprocessor.

A Quart in a Pint Pot

What actually happens is that 16K of the ROM overlays 16K of RAM. The system is arranged such that you always read the information from the ROM and not the RAM underneath it. The other 4K of ROM also overlays RAM but this ROM is invisible to the user and the 4K of RAM is available for programs.

(It is possible to switch the ROMs in and out of the memory map to gain extra RAM space. However this is not easy, or advisable, from BASIC and so we have not included this facility in this book.)

And Smaller and Smaller and Smaller ...

What we really have then, is a computer with 48K of RAM and 16K of ROM. Unfortunately, not all of this RAM is available to us for writing programs. This is because a further 6K of RAM is required by BASIC and the operating system for storing important system information. (Pointers, tables etc.)

We are now down to 42K of RAM which is available to us for programs, this is called the user RAM. However, there is more to come (or rather, go) because only 38K of the user RAM is available for writing BASIC programs. The other 4K is available to us, but it cannot be used for storing BASIC programs. Its main use is to store machine code routines (this is where the BASIC extension program in Chapter 10 is located).

The Standard Memory Map

So, the memory map for the standard BASIC configuration contains 16K ROM, 6K of system RAM, 38K of BASIC user RAM and 4K of other user RAM. Figure 1.1 shows how all this fits into the 64K memory space available.

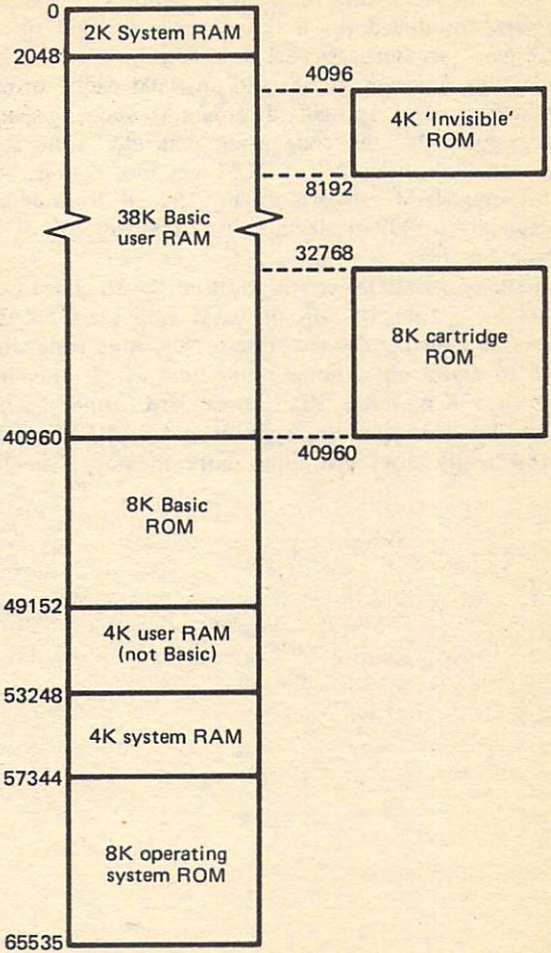

0
2K System RAM
2048
4096
4K 'Invisible' ROM
8192
38K Basic user RAM
32768
8K cartridge ROM
40960
40960
8K Basic ROM
49152
4K user RAM (not Basic)
53248
4K system RAM
57344
8K operating system ROM
65535

Fig. 1.1

One of the main advantages to having a 64K RAM machine is that you will never have to buy add-on RAM packs just so you can play the latest version of Space Zappers or whatever. Any software produced for a Commodore 64 will run on your machine – satisfying, isn't it!

It is possible however, to get add-on ROM packs, or cartridges, which contain specialised programs such as games, word processors, machine code assemblers etc. The ROM cartridges usually contain 8K of ROM which of course, overlays 8K of user RAM. This is usually the 8K from address 32768 decimal to 40959 decimal, i.e. the top 8K of the BASIC program area.

Even with an 8K ROM cartridge fitted the standard Commodore 64 has a total of 30K of RAM available for BASIC programs with a further 4K available for machine code work. Compared to many other home computers which only have 48K or even 32K of RAM fitted as standard (some of which will be used by their operating systems and BASIC) the Commodore 64 really does give you more memory for your money.

Chapter 2

RANDOM NUMBERS

Given a single toss of a normal unbiassed coin what is the chance that the coin will fall heads? Well, there are two possible outcomes of tossing the coin, (not counting the coin balancing on its edge) it can either fall heads or tails. Being an unbiassed coin each of these two events is equally likely. And so it follows, that with a lot of throws, half of the time the coin will fall heads and the other half of the time it should fall tails.

If you wanted to simulate the tossing of a coin on your computer then you would need some method of generating random numbers that fall into two equally likely groups. One group can then represent heads and the other group can represent tails. On the Commodore 64 there is a numeric function RND. The job of this function is to produce random numbers in the range 0 to 1 (where 0 can be produced but 1 can not).

Digital computers such as the Commodore 64 generally have difficulty in producing true random numbers. This is because to be truly random, the next number generated must be completely unpredictable. But a digital computer can only follow a set and predictable sequence of steps to generate the next random number given some other value as a starting point. This starting point is called a random number seed and on the Commodore 64 this seed can be provided in several ways depending on the sign of the contents of the brackets following the keyword RND. If the value in the brackets is negative then the seed value is changed in such a way that it depends on the number in the brackets after the minus sign. As an example run the following program:

```
10 PRINT RND(-1)
20 PRINT RND(-2)
30 PRINT RND(-3)
```

When you do this the answers should be as follows:

2.99196472 E–8
2.99205567 E–8
4.48217179 E–8

It may surprise you to see that we have managed to predict correctly the output from a random number generator, but in reality the function of RND with a negative argument (the numbers in the brackets after a function name are called arguments) is to generate new random number seeds from the value of the argument. This means that every time you run the little program above you will always get the same answers. If the RND argument is zero then a number is generated using the current value of the free running hardware clock called the jiffy clock which is updated at the rate of 60 counts per second. If the value in the brackets is positive then the result will just be the next value in a calculated sequence starting with the current seed value. The best strategy then, for generating random numbers is to use a zero argument all the time so that it is virtually impossible to know the value that is used as the random seed. One other important feature of the random number generator as well as being unpredictable is that every number that it can produce (i.e. 0.00000000 – to 0.99999999) has an approximately equal chance of being picked. The technical way to say this is to say that the numbers are uniformly distributed.

Returning now to our coin tossing problem, you should be able to see that all you need to do is to split the RND range (0 to 1) into two smaller equally likely groups and agree that if the generated number falls into one group you will call the result heads and if it falls into the other group you will call the result tails.

The following simple program illustrates this idea:

```
10 LET X=RND(0)
20 IF X<0.5 THEN PRINT "HEADS"
30 IF X>=0.5 THEN PRINT "TAILS"
```

Here the two smaller groups that I have chosen are 0 to 0.49999999 and 0.5 to 0.99999999. As every number in the range 0 to 1 is equally likely to be chosen and as the two groups are the same size it follows that *Heads* and *Tails* in the simulation should be random and equally likely. Run the program a few times and convince yourself that this is quite a good simulation of tossing a real unbiassed coin.

Splitting the RND range into equally sized smaller ranges to generate numbers that have a given chance of being produced is all right. But more often, what you will want to produce are random integers in a given range. For example, to simulate a dice it would be useful just to generate a random integer in the range 1 to 6 and not have to split up and test the RND range, 0 to 1. This can be done fairly easily using another numeric function as well as RND and a little maths. The extra numeric function needed is INT. As with RND the numeric expression that INT is to operate on is enclosed in brackets after the word INT. The INT function returns the value of the largest integer not greater than the value of the expression in the brackets. This sounds complicated but what it means is that the INT function rounds down to the nearest integer. So that, for example:–

```
INT(3.1) = 3
INT(4) = 4
INT(–3.1) = –4
```

To understand the last example remember that –4 is less than –3.1 so that when –3.1 is rounded down to the nearest integer you get –4 and not –3 as some of you may have expected at first glance.

Looking at throwing the dice again, if you generate a number with RND(–1) and multiply it by 6 you will get a decimal number in the range 0 to just less than 6. Using the INT function on this result will give a random integer in the range 0 to 5. To simulate a dice, you only need to add one to this result to get a random integer in the range 1 to 6 as required. This procedure is not difficult and

can all be achieved in a single line of code as follows:–

```
10 LET X=INT(RND(0)*6)+1
```

In general, random integers in any range A to B may be produced by the formula

```
10 LET X=INT(RND(0)*(B-A+1))+A
```

If A is 1, so that the required range is 1 to B, this simplifies to

```
10 LET X=INT(RND(0)*B)+1
```

You will see that if you set A=1 and B=6 you get the dice line of code back again.

A Simple Card Shuffle

Let me move on now to a more complicated situation that you may wish to simulate: generating and shuffling a pack of cards. Simple card games on a computer are fairly easy to program once you have sorted out an acceptable representation of a pack of cards and how to shuffle it. For the purposes of demonstration I am going to ask you to imagine a set of 52 empty pigeon holes, and to believe that the problem of sorting out an acceptable representation of a pack of cards could be considered to be solved if the integers between 1 and 52 could be randomly placed into the 52 pigeon holes, so that there was a different number in each pigeon hole. If each card is now given a different number (e.g. 1=ace of spades, 2=2 of spades ..... 13=King of Spades, 14=ace of clubs etc.) in the range 1 to 52, then the number in pigeon hole 1 will represent the card at the top of the shuffled deck, the number in pigeon hole 2 will represent the second card in the shuffled deck and so on.

One method of achieving this result is to start off with the first pigeon hole and generate a random number in the

range 1 to 52 which is then inserted into the pigeon hole. Next you would consider pigeon hole 2 and generate a second random number for it. There is a problem now, though, because it is possible that the same number has been generated for pigeon hole 2 that was generated for pigeon hole 1, and since there is only one of each card in a pack, we cannot use this number again. The solution to this problem is to check back to see if the number just generated is one that has already been used. If it is not then you may use it freely, otherwise it must be discarded and another number generated and checked. This procedure should be repeated for each of the 52 pigeon holes. A program using this algorithm is as shown. If you have a machine, type it in and then come back and we'll talk about it.

```
10 DIM PACK(52)
20 FOR COUNT=1 TO 52
30 R=INT(RND(0)*52)+1
40 CHECK=COUNT-1
50 GOTO 80
60 IF PACK(CHECK)=R THEN 30
70 CHECK=CHECK-1
80 IF CHECK>0 THEN 60
90 PACK(COUNT)=R
100 PRINT R;
110 NEXT COUNT
120 END
```

If you have not done much programming yet, then the operation of some of the lines of the program may need some explanation. So far in the discussion, I have mentioned pigeon holes to store the cards. These are implemented in BASIC as a set of 52 variables called an array. Line 10 of the program tells BASIC that this array is to be called PACK and that it is to be 52 variables long. (PACK(1)=pigeon hole 1, PACK(2)=pigeon hole 2.....PACK(52)=pigeon hole 52.)

Lines 20 and 110 form a loop. The contents of this loop (lines 30 to 100) will be executed once for each value of the variable COUNT (i.e. 1 to 52 as per line 20). Line 30 generates the random numbers which represent the cards. Lines 40 to 80 check back over the previously generated numbers to see if the latest number has already been used. If it has, a jump is made back to line 30 to generate a new random number, otherwise the program continues on with line 90. This line enters the new number into the PACK. Finally, line 100 prints the latest card number generated onto the screen. If you entered the program, run it a few times and convince yourself that the list of numbers is printed in random order and that each number appears only once.

You will notice if you run the program that it starts off fast enough, but as more numbers are generated, the program begins to slow down so eventually the last few numbers can take many seconds to appear. This is because as more and more numbers are generated, the chance of rejecting a generated number because it has been used before increases slowly until, for the last few numbers, most of the numbers generated are being checked and rejected. This means that hundreds of random numbers are being generated and then not used. To get a feel for the size of the waste, add the following lines of code to the previous program, and run it a few more times.

```
15 WASTE=-52
35 WASTE=WASTE+1
115 PRINT "NUMBERS WASTED =";WASTE
```

Here line 35 increments the variable WASTE each time a number is generated and line 115 prints this total at the end. Line 15 sets the variable WASTE to an initial value. You might have expected this to be zero, and it would be if you needed a total count of all the numbers generated. But 52 of the numbers generated are used to fill the array, and so that these are not counted, WASTE is given an initial value of −52.

This algorithm for generating a pack works, but due to the wasted effort, it is not very efficient. What is needed is an algorithm with less waste. There is one, and it is set out in the following program:—

```
10 DIM PACK(52)
20 FOR COUNT=1 TO 52
30 PACK(COUNT)=COUNT
40 NEXT COUNT
50 FOR COUNT=1 TO 52
60 R=INT(RND(0)*52)+1
70 TEMP=PACK(R)
80 PACK(R)=PACK(COUNT)
90 PACK(COUNT)=TEMP
100 NEXT COUNT
110 FOR COUNT=1 TO 52
120 PRINT PACK(COUNT);
130 NEXT COUNT
140 END
```

Again line 10 tells BASIC to create an array of 52 variables called PACK. Lines 20 to 40 set up the array so that PACK(1)=1, PACK(2)=2, ... PACK(52)=52. This in effect lays the pack out in order. Lines 50 to 100 perform the shuffle itself. This is done by taking each card position in the pack in turn and exchanging the card number found there with the card number contained in a randomly generated card position. By the time every card position in the array has been exchanged in this way, the pack is completely randomly shuffled. During this process, only 52 random numbers are generated and all of them are used regardless of their values. Finally, lines 110 to 130 print out the numbers in the PACK array. Enter this program and run it a few times to convince yourself that the result is indeed random.

This shuffle algorithm is very efficient and consequently

it executes appreciably faster than the first one we looked at, as you can verify if you note the run times of the two programs.

Playing your Cards Right

We will now look at a simple card game program which is very loosely based on a TV game show you may have seen.

The pack is shuffled and the top card displayed. You then have to guess whether the next card will have a higher or lower value than the previous one. This guessing process is repeated and a total is kept of the number of correct guesses. The game is repeated when an incorrect guess is made and a best score so far is displayed. The game is simplified to make it easier to understand its operation. In particular, the cards are printed on the screen as ACE, 2, 3, ... QUEEN, KING with no regard for suit. In addition, no check is made to see if the pack runs out, i.e. no check is made to see if you have made 51 correct guesses in a row and have thus run out of cards. These are things you might consider adding to this program yourself. The second card shuffling algorithm is the one used in this game.

```
10 DIM PACK(52)
20 BEST=0
30 CARD=1
40 FOR COUNT=1 TO 52
50 PACK(COUNT)=COUNT
60 NEXT COUNT
70 FOR COUNT=1 TO 52
80 R=INT(RND(0)*52)+1
90 TEMP=PACK(R)
100 PACK(R)=PACK(COUNT)
110 PACK(COUNT)=TEMP
120 NEXT COUNT
130 PRINT "{clr/home}"
```

```
140 PRINT "BEST SCORE SO FAR =";BEST
150 PRINT
160 PRINT "YOUR CARD IS ";
170 C=INT(PACK(CARD)/4+0.75)
180 GOSUB 350 : PRINT C$
190 TEMP=C
200 PRINT "WILL THE NEXT CARD BE "
210 PRINT "HIGHER OR LOWER (H/L)";
220 INPUT G$
230 CARD=CARD+1
240 C=INT(PACK(CARD)/4+0.75)
250 IF G$="H" THEN 330
260 IF G$<>"L" THEN 200
270 IF C<TEMP THEN 150
280 GOSUB 350
290 PRINT "NO, NEXT CARD IS ";C$
300 CARD=CARD-2
310 IF CARD>BEST THEN BEST=CARD
320 GOTO 30
330 IF C>TEMP THEN 150
340 GOTO 280
350 C$=STR$(C)
360 IF C=1 THEN C$="ACE"
370 IF C=11 THEN C$="JACK"
380 IF C=12 THEN C$="QUEEN"
390 IF C=13 THEN C$="KING"
400 RETURN
```

This program is fairly straightforward, and I leave you to play it and sort out how it works.

Chapter 3

PEEK AND POKE

A lot of people treat the PEEK and POKE keywords as though they had some mystical power. Consequently they fight shy of using them in their own programs. This fear stems from the problem that although PEEK and POKE are very powerful and flexible facilities they convey no information about their actions when written in a BASIC program.

For example, on the Commodore 64 a POKE 53272, 23 tells you nothing about its action. (It actually switches the character set to lower case.) Things get even worse if the PEEK or POKE values are variables. I mean, what on earth does a POKE AD,S% do?

The only way to remove this mystical appearance of PEEK and POKE is to clearly document their actions within your programs. This is particularly important on the Commodore 64 because most of the sound and graphics facilities are controlled by PEEKs and POKEs.

PEEK and POKE Revealed

So, exactly what do PEEK and POKE do? Well, perhaps surprisingly, their actions are very simple:–

PEEK reads the value in a specified memory location. It is a BASIC function because it returns a value.

POKE writes a specified value into a specified memory location. It is a BASIC statement because it performs some action.

The correct form, or syntax, of the PEEK function is:–

PEEK (memory location) e.g. PEEK (53272)

The memory location parameter can only take values which are decimal integers between 0 and 65535. This is because there is only 64K (65536 bytes) of memory available. Since all the memory locations are only 8 bits (or 1 byte) wide, PEEK can only return values between 0 and 255.

The correct syntax of the POKE statement is:–

POKE memory location, value e.g. POKE 53272,23

As for the PEEK function, the memory location parameter can only take decimal integer values between 0 and 65535. Similarly, the value parameter can only take decimal integer values between 0 and 255.

Using PEEK and POKE on the Commodore 64

This little section could fill a book all on its own! There are 65536 memory locations on the Commodore 64 and you can PEEK values from and POKE values to most of them. (It is not possible to POKE new values into Read Only Memory areas like the BASIC interpreter at address 40960 to 49151.) POKEing values into most addresses is usually not particularly rewarding (the machine just "hangs"). Some special locations however, can be used to good advantage.

We have already seen that POKE 53272, 23 switches to lower case. In addition, POKE 53272, 21 switches back to upper case. These two POKEs can be very useful for making text based programs look much neater.

It can often be an advantage to POKE directly to the screen without having to worry about moving the cursor to the right place and then PRINTing. The screen is mapped into 1000 bytes of memory from 1024 decimal up to 2023 decimal. Each value between 0 and 255 will print a different character. (Although some are control codes like RETURN and DELETE.) The position on the screen depends on which screen memory address you POKE into.

For example 1024 decimal is the top left corner. To print on, say, the second space of the next line down (41 characters

away from the top left corner) you should POKE 1065, value.

If you like, you can use two variables X and Y to point to the screen locations for you. X must remain in the range 0 to 39 and Y must remain in the range 0 to 24. Any location on screen can then be calculated from LOCATION=Y*40+X+1024.

When you are POKEing to the screen you also have to remember to POKE the appropriate character colour into the colour memory. (The BASIC PRINT statement does this for you.)

Colour memory runs from 55296 up to 56295. This gives 1000 bytes which directly map onto the screen memory. For example, the colour of the character in screen memory location 1065 is contained in colour memory location 55337. (Note that although each colour memory location can have 256 different values there are only 16 colours available, so that only the values 0 to 15 have any meaning.)

So, to put say, the A character at some X,Y location in, say, red:–

```
POKE Y*40+X+1024,1 : POKE Y*40+X+55296,2
```

Note, the character code for A is 1 and the colour code for red is 2.

Two other useful screen colour locations are 53280, which controls the border colour, and 53281 which controls the screen background colour.

For example, POKE 53280,0 : POKE 53281,0 sets the whole screen background to black (which we find much more restful!).

Finally the three registers at address 160 to 162 are used as a single 24 bit counter. The value in this super register is incremented automatically every 1/60th of a second. This is the real time jiffy clock mentioned in Chapter 2.

This clock can be used as an event timer by resetting the whole register to 0 and then PEEKing the value after some event has occurred. (Because it takes time to PEEK the values the time obtained will only be approximate.)

For example, the following short program will print the time taken between the question being asked and the user entering his answer. This may be useful in educational type programs.

```
10 PRINT "THIS IS A QUESTION"
20 POKE 162,0
30 POKE 161,0
40 POKE 160,0
50 INPUT "WHAT IS THE ANSWER ";A$
60 T0=PEEK(162)
70 T1=PEEK(161)*256
80 T2=PEEK(160)*65536
90 PRINT "TIME TO ANSWER = ";
100 PRINT (T0+T1+T2)/60;" SECONDS"
```

Lines 20–40 zero the jiffy clock.

Lines 60–80 PEEK the three jiffy clock bytes after the INPUT has been completed. The multiplication factors are required to convert the three individual register values into one 24 bit register value.

A Bit at a Time

So far we have been POKEing values into registers with no thought for the value that was already there. It is often the case that you want to set (set = 1) a bit within a register without changing any of the others. This is because the Commodore 64, like most other machines, uses individual bits to switch facilities on or off, so that one register may control eight different facilities.

For example, we have already seen that POKE 53272,23 switches lower case on and that POKE 53272,21 switches upper case on. At first sight there doesn't seem to be any relationship between the numbers 21 and 23. But when they are converted from decimal to binary (binary shows the state of each of the bits) the difference becomes obvious:–

21 decimal = 00010101
23 decimal = 00010111

The last but one bit on the right hand side is the only difference between them. To make describing the bits easier they are numbered from 0 to 7 going from right to left. So we can say that bit 1 is the bit within register 53272 which switches between upper and lower case. If bit 1 is set then lower case is selected. If bit 1 is reset then upper case is selected.

To be able to sort out which bits are set for any given decimal number you need to be able to convert from decimal to binary and back. The conversion chart in Figure 3.1 should make this task much easier.

To use the chart to convert from binary to decimal, you read the high 4 bits (bits 4–7) along the top and the low 4 bits (bits 0–3) down the side. Where they cross in the chart is the decimal equivalent.

To convert from decimal to binary, you first find the decimal number in the body of the chart, and then read off the high 4 bits (bits 4–7) from the top line and the low 4 bits (bits 0–3) from the side.

Suppose you now want to set bit 1 of register 53272 (to switch to lower case) without disturbing the other bits and without knowing exactly what value is already stored there.

The obvious solution is to PEEK the value of register 53272, set bit 1 and then POKE the new value back. But how do we set bit 1 without disturbing the others?

The answer is, by using two logical functions called AND and OR.

These two functions operate by comparing two 8 bit values on a bit for bit basis and producing a result based on these comparisons.

The AND function puts a 1 in the result if both bits, in the values being compared, are 1.

The OR function puts a 1 in the result if either bit, in the values being compared, is 1.

For example, if we use the OR function to OR the

MSB LSB	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0001	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0010	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
0011	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0100	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
0101	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
0110	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
0111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
1000	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
1001	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
1010	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
1011	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
1100	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
1101	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
1110	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
1111	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Fig. 3.1

decimal values 171 and 64 we get the following result:–

171 decimal	= 1 0 1 0 1 0 1 1	If there is a 1 here
64 decimal	= 0 1 0 0 0 0 0 0	OR a 1 here
Result (235 decimal)	= 1 1 1 0 1 0 1 1	there will be a 1 here.

If we now use the AND function to AND the decimal values 235 and 191 we get the following result:–

235 decimal	= 1 1 1 0 1 0 1 1	Only if there is 1 here
191 decimal	= 1 0 1 1 1 1 1 1	AND a 1 here
Result (171 decimal)	= 1 0 1 0 1 0 1 1	will there be a 1 here.

Now, to return to the problem of setting bit 1 of register 53272. What we need to do is to find a binary number which has only bit 1 set and then OR it with the existing contents of register 53272.

If you look back at the binary to decimal conversion chart you can see that the number we require (00000010) is 2 in decimal. The procedure now is to PEEK (53272), OR this value with 2 (to set bit 1) and then POKE this value back into location 53272. This can all be done in one line of BASIC:–

```
POKE 53272,PEEK(53272) OR 2
```

Notice that the value parameter of the POKE statement takes the value of PEEK (53272) OR 2 which is the existing value of register 53272 with bit 1 set.

Would this process have worked if bit 1 had already been set? Try it on paper and see. You will find that this process always works for any bit, or group of bits, regardless of whether they are already set.

Now suppose we want to reset bit 1 of register 53272 (to switch to upper case) without disturbing the other bits and without knowing its existing contents.

The solution is very similar to the last problem. The difference is that we now need to find a binary number which

has all bits except bit 1 set and then AND it with the existing contents of register 53272.

If you look at the conversion chart again you can see that the number we require (11111101) is 253 in decimal.

The procedure now is to PEEK(53272), AND it with 253 and then POKE this value back into location 53272. This can be done in one line of BASIC:–

```
POKE 53272,PEEK (53272) AND 253
```

Notice that the value parameter of the POKE statement takes the value of PEEK (53272) AND 253 which is the existing value of register 53272 with bit 1 reset.

Would this procedure have worked if bit 1 had already been reset? If you try it you will find that it always works for any bit, or group of bits, regardless of whether they were already reset.

The two numbers which were specially selected for their bit patterns (2 and 253) in these problems are called masks. This is because they mask in or out specific bits, or groups of bits. This technique of bit masking will be invaluable when we come to look at the Sprite, sound and high resolution graphics facilities in the next chapters.

Chapter 4

CHARACTER GRAPHICS

The Commodore 64 has 2 completely different display modes: High resolution and Low resolution. This chapter will deal with the second of these in more detail, the other being dealt with in Chapter 7.

In Low resolution, the screen is composed of 1000 character locations laid out as 25 lines of 40 characters. All the characters that can be displayed normally on the Low resolution screen are contained in ROM in two separate character sets. These two character sets are only available one at a time and they can be switched between in direct mode by simultaneously pressing shift and the Commodore key. Character set 1 contains the upper case characters, the numbers, some special characters used within BASIC and a set of useful graphics characters. This is the character set that is in use when the machine is first switched on. The second character set contains just a lower case character set and an upper case character set. There are two common ways to put characters onto the low resolution screen from within BASIC and we will examine them both in this chapter.

The first of the two is the PRINT statement. Using a function that we shall examine in a moment, the PRINT statement is capable of producing some fairly impressive graphics and simple animation. In order to produce animation, some facility is needed to enable the program to control where on the screen the next character is to be printed. On the Commodore 64 this is achieved by inserting cursor movement control characters into the string that is being printed. The function used for this purpose is a string function called CHR$.

CHR$

CHR$(X) is a string function whose purpose is to return a single character string which is the character whose ASCII code (American Standard Code for Information Interchange) is the value of the expression contained in the brackets. In this case, the value of X. The contents of the brackets must be in the range 0 to 255 or an illegal quantity error message will result. The standard ASCII code only contains 128 characters numbered 0 to 127. The first 32 are special control characters, most of which control printers and terminals with functions like line feed, form feed and carriage return. The second 32 characters contain special characters used in BASIC like $,( ),+,* etc. and the digits 0 to 9. The third set of 32 characters contains the upper case letters and the fourth set contains lower case letters. On the Commodore 64 this is slightly different. The second and third sets of 32 characters still contain the special characters, the digits and the upper case letters, but there the similarity ends. The fourth set of 32 now contains a set of graphics characters instead of lower case letters and the first set of 32 characters now contains some special control codes that we are particularly interested in. In addition to this character codes 128 to 255 are also available on the Commodore 64 of which the first 32 are also of special interest. We will now take a look at a simple bouncing ball program and for this we will need the following special control codes to use in the CHR$ function.

CHR$(X)	*Effect*
17	Cursor down
29	Cursor right
145	Cursor up
147	Clear screen and home cursor
157	Cursor left

In addition to these we will also use CHR$(113) which will print a small ball character from the graphics part of the character set.

```
10 X=10 : Y=10
20 DX=1 : DY=1
30 PRINT CHR$(147);
40 FOR C=1 TO X : PRINT CHR$(29); :
   NEXT C
50 FOR C=1 TO Y : PRINT CHR$(17); :
   NEXT C
60 PRINT CHR$(157);" ";CHR$(157);
70 IF DX=1 THEN PRINT CHR$(29); :
   GOTO 90
80 PRINT CHR$(157);
90 IF DY=1 THEN PRINT CHR$(17); :
   GOTO 110
100 PRINT CHR$(145);
110 PRINT CHR$(113);
120 X=X+DX : Y=Y+DY
130 IF X<2 OR X>39 THEN DX=-DX
140 IF Y<2 OR Y>24 THEN DY=-DY
150 GOTO 60
```

If you type this program in and execute it with RUN you will see a small ball bouncing diagonally all round the screen. A simple explanation of the operation of this program is as follows:–

X and Y are the two variables that keep track of the current position of the ball and line 10 starts these off at 10,10. DX and DY are the two variables that keep track of the direction that the ball is travelling in. If DX is positive then the ball is moving right; otherwise it is moving left: if DY is positive then the ball is moving down; otherwise it is moving up. Line 20 starts the ball moving diagonally down and to the right. Line 30 clears the screen and positions the cursor (which is where the next character printed will go) at the top left character position on the screen. Lines 40 and 50 move the

cursor, using the down and right cursor movement control codes, so that it is sitting at the position specified in variables X and Y.

Line 60 erases the ball from its previous position in preparation for reprinting it somewhere else to make it appear to move. It works by moving the cursor back to the position of the ball then printing a space (denoted by " " in the program) to erase the ball character and then moving the cursor back again to the current ball position in preparation for calculating its new location.

Lines 70 and 80 move the cursor either left or right depending on the value of DX.

Lines 90 and 100 move the cursor either up or down depending on the value of DY.

Line 110 prints the ball character at its new position.

Line 120 updates the X and Y values to show the new position of the ball.

Lines 130 and 140 check the values of X and Y to see if the ball has hit the boundary and if it has, to update the value of DX and/or DY accordingly.

Finally, line 150 causes the program to loop back to line 60 to erase the ball and move it to its next position.

As you can see, the bouncing ball program above could form the basis of many bat and ball type games, and we will see an example of such a program in the next chapter.

Now we must move on and take a look at some more of the special control codes that can be used inside the brackets with CHR$(X).

Colour

Apart from cursor movement control the next most important control codes are those that control colour. On the Commodore 64 there are 16 different shades of colour that can be used. Of these, 8 are available as special control codes. The table on page 26 lists the 8 colours and their respective control codes.

Colour	*Code*
WHITE	5
RED	28
GREEN	30
BLUE	31
BLACK	144
PURPLE	156
YELLOW	158
CYAN	159

For a demonstration of the colour control codes in use type in and run the following short program:–

```
10 RESTORE : POKE 53281,11
20 FOR C=1 TO 8
30 READ CC
40 PRINT CHR$(CC);"RAINBOW COLOUR
   DEMONSTRATION"
50 NEXT C
60 DATA 144,5,28,159,156,30,31,158
```

If you run this, you may wonder where the background colour came from. Well, it was produced by the POKE statement in line 10. As mentioned earlier, the Commodore 64 can produce 16 different shades of colour, but of these, only 8 are available as control codes to be used with CHR$. The way to access the other colours is to POKE their values into an area of the memory map which is set aside for just this purpose. Within the memory map there is an area of memory from location 55296 to location 56295 which is used to determine the colours of the characters printed on the screen. As you can see, this is exactly 1000 bytes long, which is also just the number of character positions (25 x 40 = 1000) that there are on the screen. This is because there is one colour byte for each position on the screen. This means that the colour can be specified independently for each

character position on screen. Within the 1000 byte colour map the bytes are laid out as follows:–

Byte 55296 is the colour byte for the top left location on screen. The colour bytes are then used, one for each screen position left to right across the top line of the screen so that byte 55335 is the colour byte for the rightmost character position on the top line. Byte 55336 is the first byte used at the left hand end of line 2 and then across line 2, left to right to byte 55375 and so on until the right hand end of the bottom line is reached at byte 56295. What is really needed now, if this information is to be useful, is a method of finding the colour byte number given the X and Y location on the screen of the character position that we wish to change the colour of. For the purposes of all such expressions, it is always easier if the X and Y co-ordinates count from 0. This means that on the Commodore 64 screen the X co-ordinates would range from 0 to 39 and the Y co-ordinates from 0 to 24. Given these values for X and Y then an expression for the colour map byte would be: MB=55296+40*Y+X.

The correctness of this expression can easily be seen if it is considered as follows:–

The map byte is calculated as the address of the first byte (55296), plus Y complete lines of 40 characters down the screen, plus X characters into the required line. The values to be poked into the colour map byte locations are given in the table, shown below for completeness.

0=Black	8=Orange
1=White	9=Brown
2=Red	10=Light Red
3=Cyan	11=Grey 1
4=Purple	12=Grey 2
5=Green	13=Light Green
6=Blue	14=Light Blue
7=Yellow	15=Grey 3

We now know how to change the colour that any character will be printed in, but what about the background colour?

Well, as you saw in the small program we used to demonstrate colour earlier, the background colour for the whole screen is controlled by the contents of memory location 53281. This location can be poked with any of the 16 colour values just as the colour memory can, and when this is done the background colour for the whole screen will change to the colour selected. Finally, in addition to the location for changing the background colour there is also a location for changing the colour displayed in the border. This is at location 53280. So, for example, to change the border colour to orange, you would enter POKE 53280,8.

Chapter 5

GET

In this chapter we will expand on those things that we have just seen on character graphics in Chapter 4, by learning how to scan the keyboard and a new and faster method of controlling the bouncing ball. From these we shall develop a simple squash type bat and ball game controlled by the function keys on the keyboard.

INPUT Routines

When you see an INPUT statement in a line of program how often do you stop to consider just how much work is done by it? Never, I suspect, and yet the algorithm for an INPUT statement is fairly complex and it may not always be just the one you want. Consider the following flow chart (Figure 5.1) which represents some of the work done by INPUT.

As you can see, there is a fair amount of work to be done, scanning the keyboard, putting the characters on the screen, checking for backspace, making sure that the number of characters input does not exceed the size of the input buffer and looking for the return key to be pressed to end the input. All the things mentioned here are shown on the flowchart Figure 5.1, but this is not all that INPUT has to do. The box at the bottom of the flowchart is just labelled "Rest of Input" and this includes things like: checking that the number of entries input (separated by commas) is the same as the number of variables specified in the INPUT statement. Checking that the types of input are correct for the variables specified, e.g. checking that a number and not a string was entered when a number was required. Assigning the values entered to the specified variables and finally giving warning and error messages if any of these conditions are not met.

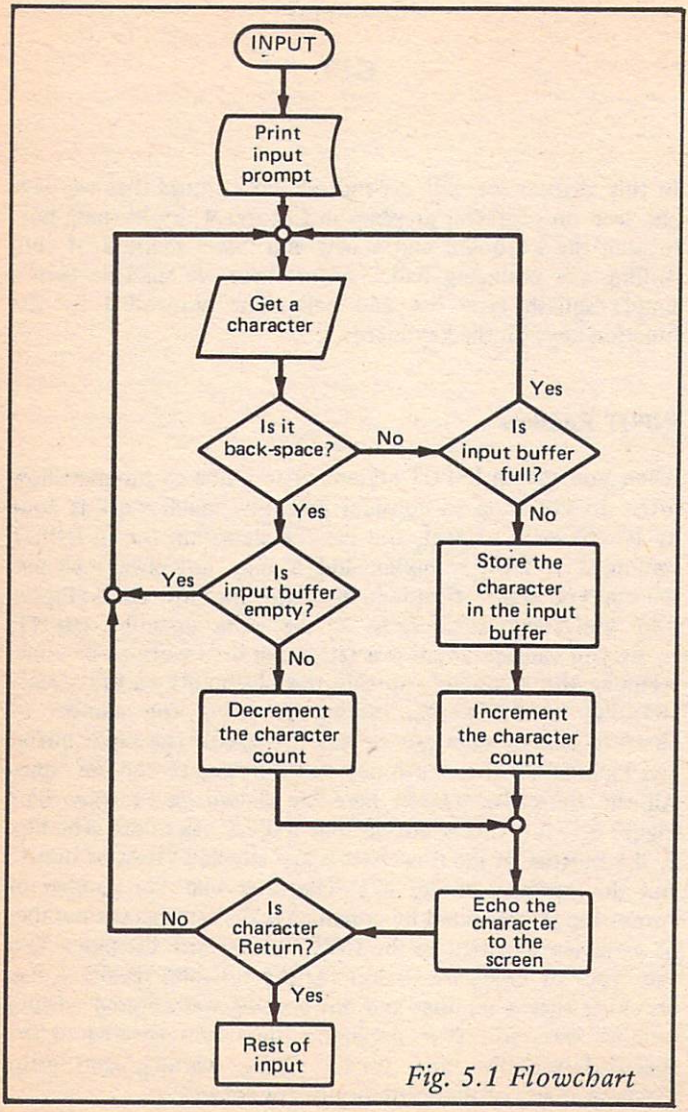


Fig. 5.1 Flowchart

As you can see, the algorithm for INPUT is fairly complex and it does its job well. But for certain tasks like controlling the bat in a graphics bat and ball game it is totally useless. The big problem with INPUT for our games application is that it needs to see the return Key pressed. This means that if we use INPUT in our graphics games then the action will freeze every time the program expects some entry to be made by the player. Obviously this is totally unacceptable and if INPUT was the only keyboard entry facility available on the Commodore 64 then graphics games needing keyboard entries would be impossible to produce. However, as you have probably guessed from the title of this chapter, there is another statement available in the BASIC that can be used to take entries from the keyboard and its algorithm is somewhat simpler and does not require a return key press at the end. This new statement is called GET.

Keyboard Scan

On the Commodore 64 there is a special buffer which is 10 bytes long called the type ahead buffer. This buffer sits in memory and occupies memory locations 631 to 640 inclusive. When any key is pressed on the keyboard a code for the key is stored at memory location 197. From here the ASCII code for the key pressed is passed into the type ahead buffer. If a second key is pressed before the machine has had time to deal with the first character then the new character will not be lost but will be stored in the type ahead buffer next to the first. In addition to this the contents of memory location 198 contain a count of the number of characters in the type ahead buffer still to be dealt with. If the input of keyed entries was not dealt with in this way it would be possible to lose data whenever the computer was not ready to take the next key pressed. The GET statement comes into the story at this point because it removes the character from the first position in the type ahead buffer and assigns it to the specified variable. For example GET A$ would get the character at the top of the type ahead buffer and store that character in A$. If the

type ahead buffer was empty then A$ would be assigned a value of null. You will notice that nowhere in this description of GET is there any mention of pressing Return. This is because Return is not needed to end a GET statement which in turn means that GET does not stop and wait for an input, it just takes the first character in passing or reports the fact that no key was pressed. This sounds like it is just what is needed to control a bat in a bat and ball game, but as we shall see in a moment there is still one small problem.

Try entering the following short program:

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

If you RUN this program you will find that the symbol (letter or number) on any key that you press is echoed to the screen and that this is repeated each time a key is pressed. Let us suppose you press and hold the letter A on the keyboard. The program would then operate as follows.

Line 10 would get the letter A from the type ahead buffer and store it in A$. The test in line 20 would fail because A$ is not null so the next line to run would be line 30 which would print the letter A on the screen.

Line 40 then makes the program loop back to line 10. This means that the GET statement will be executed again. What will the value of A$ be now? Well we know that the letter A does not get printed again unless we release the key and press it again. This means that the program is looping round lines 10 and 20 continuously so that we don't execute the print statement again. The only way that this can happen is if A$ has the value null so that the test in the IF THEN statement in line 20 passes and we go back to line 10. Now obviously A$ can only be null if that is the value returned by GET. This means that having once seen a key

pressed GET does not see it again no matter how long you hold the key down. This could cause a problem in controlling a bat where ideally holding a bat movement key down would cause the bat to move for as long as the key was held. The solution to this problem lies in location 197 which as we said earlier is where a code for the key pressed is first stored. It turns out that the Commodore 64 does in fact continue to see any key pressed but if the code generated is already stored at location 197 then the machine assumes (quite correctly) that it has already seen this key press and so ignores this latest data. This suggests that all we need to do is to change the contents of location 197 after executing a GET statement to fool the machine into thinking that it is seeing a different key press each time it does a GET. We can test this out by adding a little extra code to the last program to give the following:

```
10 GET A$
15 POKE 197,64
20 IF A$="" THEN 10
30 PRINT A$;
40 GOTO 10
```

Now you will see that when you press a key it continues to be printed until you release it again. Incidentally, the value 64 is used to POKE into location 197 because this is the value that the Commodore 64 stores there when no key is pressed.

As we suggested at the start of this chapter, we will be using some of the function keys to control the bat in our squash game and so this seems a good time to look at how to use these keys. In fact, it is very simple; the function keys can be scanned just like any other keys on the keyboard and they return ASCII values in the range 133 to 140 as follows:–

Function key	*ASCII values*
1	133
2	137
3	134
4	138
5	135
6	139
7	136
8	140

This means that we can use GET to store a character into a string variable and then compare this with the ASCII values from the table above using CHR$ as follows:–

```
10 GET A$ : IF A$="" THEN 10
20 IF A$=CHR$(133) THEN PRINT "F1"
30 IF A$=CHR$(137) THEN PRINT "F2"
40 IF A$=CHR$(134) THEN PRINT "F3"
50 IF A$=CHR$(138) THEN PRINT "F4"
60 IF A$=CHR$(135) THEN PRINT "F5"
70 IF A$=CHR$(139) THEN PRINT "F6"
80 IF A$=CHR$(136) THEN PRINT "F7"
90 IF A$=CHR$(140) THEN PRINT "F8"
100 GOTO 10
```

When we are armed with one further piece of information we shall be ready to tackle the game. You may remember how, in the last chapter, we used cursor control codes in PRINT statements to move a ball around the screen. This worked quite well but with the extra program involved in scanning the keyboard, maintaining a score board and moving a bat around the screen the techniques we have learned so far turn out to be a little too slow.

At the end of the last chapter we looked at how the colour is stored and how to access it with the POKE statement.

Well, the character screen itself is laid out in a similar way with 1000 bytes from 1024 to 2023 inclusive being used to store codes for the characters that appear in the 1000 locations on the screen. These 1000 bytes are laid out on the screen in exactly the same order as the bytes in the colour memory map were and so we can use a similar expression to the one we used on the colour map to access a character byte given its X and Y co-ordinate. In this case the expression is CB=1024+Y*40+X which is different from the colour map byte expression MB=55296+Y*40+X only in the first term. And this first term only determines the starting place for the 1000 byte block of memory.

A list of the character codes that we will be using to POKE into the character memory is as follows:

Code	*Character*
48–57	Digits 0–9 for BALL count
81	BALL
32	space
160	Bat (3 used for whole bat)

Anyone for Tennis?

The listing for a simple bat and ball game is given below. In play, the bat is controlled by two of the function keys. Function key 1 moves the bat up and function key 7 moves the bat down. You have 8 balls to play and each time you hit a ball your score is increased by 1.

You should see, as you type in and use the game, that it uses most of the techniques and ideas that we have seen and developed in the last two chapters.

```
10 POKE 53281,6 : POKE 56307,7
20 PRINT CHR$(147);CHR$(5);
30 PRINT "      BALL      HITS 0    "
40 HITS=0 : BAT=10 : NXT=10
```

continued

```
50 FOR BL=1 TO 8
60 POKE 1035,BL+48
70 X=28 : Y=INT(RND(1)*20)+3
80 DX=-1 : DY=-1
90 LOC=1024+Y*40+X : COL=LOC+54272
100 POKE LOC,81 : POKE COL,7
110 GET A$ : POKE 197,64 :
    IF A$="" THEN 170
120 IF A$=CHR$(133) AND BAT>1 THEN
    NXT=BAT-1
130 IF A$=CHR$(136) AND BAT<22 THEN
    NXT=BAT+1
140 FOR C=BAT TO BAT+2
150 POKE 1054+C*40,32
160 NEXT C
170 FOR C=NXT TO NXT+2
180 POKE 1054+C*40,160
190 POKE 55326+C*40,7
200 NEXT C
210 BAT=NXT
220 IF PEEK(LOC)<>160 THEN 250
230 DX=-DX : HITS=HITS+1
240 PRINT CHR$(19),,HITS
250 POKE LOC,32
260 X=X+DX
270 Y=Y+DY
280 IF X>36 THEN 320
290 IF X<2 THEN DX=-DX
300 IF Y<3 OR Y>23 THEN DY=-DY
310 GOTO 90
320 FOR Z=1 TO 1000 : NEXT Z :
    NEXT BL
330 PRINT CHR$(147);
340 PRINT "YOU SCORED";HITS;
    " POINTS"
```

We leave you to sort out how this program works while we gather our thoughts in preparation for the more advanced things yet to come.

Chapter 6

SPRITE GRAPHICS

Sprites are a fairly new development in home computers, so it's probably worth spending a little time finding out exactly what they are and what they can do.

The simplest model for a Sprite is to think of it as a small, separate, high resolution screen, which can be overlaid on the main screen anywhere you like. An immediate advantage that we can see is that moving the Sprite around the main screen is fairly simple, it is only necessary to specify the new position for the Sprite screen, we don't need to worry about moving the individual dots which make up the shape of the Sprite. In addition, it is not necessary to redraw the bits of the main screen which were overlaid by the Sprite, because the Sprite never actually existed on the main screen. When we move the Sprite the main screen remains unaltered.

The Commodore 64's BASIC can control up to eight Sprites at the same time. Using our model of a Sprite you can see that, sooner or later, some of the Sprite screens are going to overlay each other. The Sprite graphics system allows us to detect these collisions between Sprites and also to specify how they will collide, or rather how they will overlay each other. By overlaying Sprites in a specific order it is possible to create some very interesting 3D effects. And using the collision detection facilities of Sprites enables some very exciting games to be implemented fairly easily.

So, in general terms, a Sprite is a very flexible graphics object with a lot of built in intelligence. By using Sprites some very impressive graphics programs can be implemented using fairly short BASIC routines. The price that you have to pay for this intelligence is that, for each Sprite, a lot of setting up needs to be done before the Sprite can be used.

Creating Sprites

Our model of Sprites describes them as small, separate, high resolution screens. In reality these screens are 24 dots wide by 21 dots high. The dot spacing on the Sprite screen is exactly the same as that on the main high resolution screen, so that a Sprite can exactly overlay any portion of it. These Sprite screens are controlled by a special chip in the Commodore 64 called the Video Interface Controller, or VIC chip for short. To set up a Sprite we have to tell the VIC chip all about the Sprites attributes, including its shape, colour, position on the main screen etc.

Obviously, the first thing to do when setting up a Sprite is to design its shape. Since it has to fit on a 24 x 21 dots screen, the simplest way is to get hold of some squared paper (about 5 mm squares are best) and draw a 24 x 21 squares box on it to represent the Sprite screen. Each square on the paper now represents a dot on the Sprite screen. To actually draw the Sprite it is best to begin by sketching in the outline of the shape you require, and then fill in those squares which must be lit in order to produce the shape. Figure 6.1 shows this method of Sprite design.

We now have, on paper, a representation of the Sprite screen. This must be converted into a form which can be stored in the computer. Since the basic unit of storage is a byte (8 bits) the Sprite design must be converted into a string of bytes.

This can be done by treating each square as an individual bit. They can then be grouped together in groups of 8, starting at the top left hand corner and working across and down the Sprite design, to produce a string of 63 bytes. Figure 6.2 shows the order in which the squares are grouped together.

We now have 63 bytes, each one containing some filled in squares and some empty squares (some may be all empty and some may be all filled in of course) which must be converted to numbers so that they can be stored in memory.

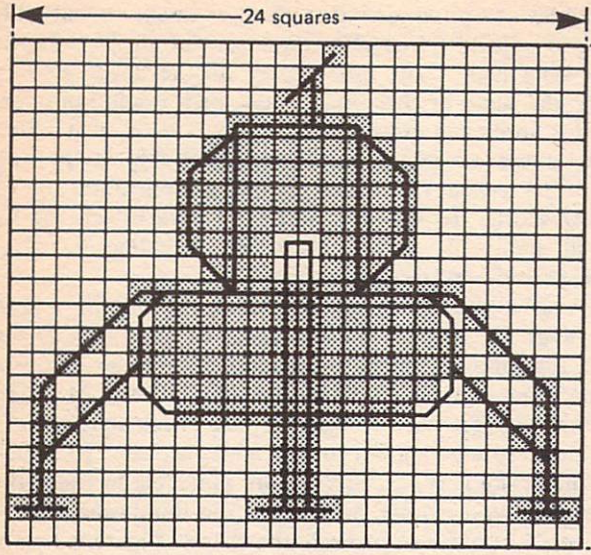


Fig. 6.1

If we give each filled in square the value 1 and each empty square the value 0 we end up with an 8 bit binary number. This must now be converted to a decimal number so that we can use the BASIC POKE statement to poke the individual bytes into memory. Figure 3.1 in Chapter 3 can be used to convert the binary numbers to decimal.

Locating Sprite Data

We now have 63 bytes of Sprite data which must be stored somewhere in memory – the question is where? In order to

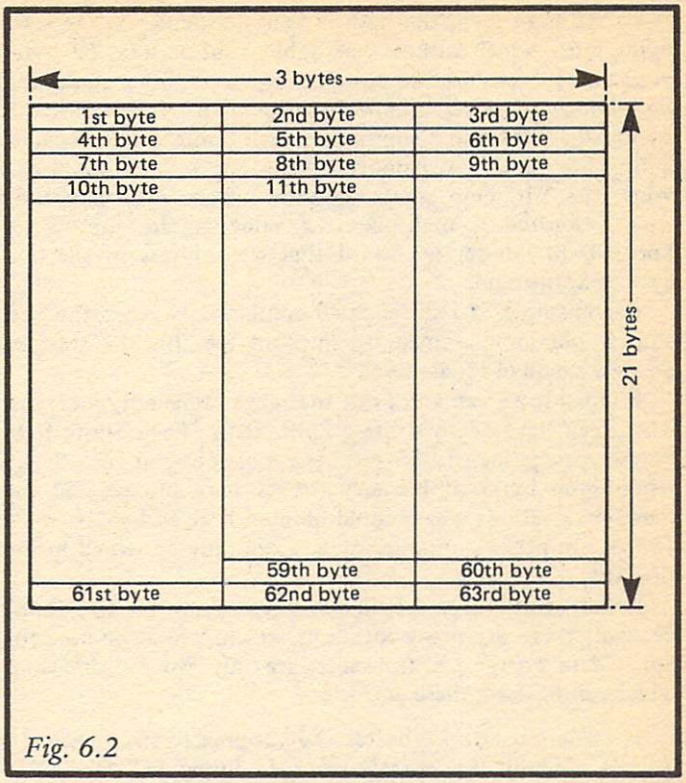


Fig. 6.2

understand where we can safely store Sprite data we need to look at how the VIC chip sorts out where the data for each Sprite is stored.

The VIC chip can control up to 8 Sprites from BASIC, each with different data. To sort out where the data for each Sprite is located it looks at a group of memory locations (or registers) called the Sprite Data Pointers. These are at locations 2040 decimal to 2047 decimal. Each register points to the data for one Sprite, for example, 2040 points to Sprite 0's data, 2041 points to Sprite 1's data and so on.

As we have seen, the data for any particular Sprite is 63 bytes long, which means that a block of at least 63 bytes of memory is required to store it. However, for a computer, the number 64 is much more convenient ($64 = 2^6$). For this reason the VIC chip assumes that each Sprite's data occupies a 64 byte block of memory. (The last byte is not used.) When the VIC chip wants to know where a given Sprite's data is located it multiplies the value in the appropriate Sprite Data Pointer by 64 to find the address of the first byte of Sprite data.

For example, if register 2040 contained 14, then the VIC chip would look at memory location 896 for the start of Sprite 0's data (14*64 = 896).

From this we can work out the range of memory locations which can be used for storing Sprite data. Each Sprite Data Pointer can point to 256 different locations and each Sprite requires 64 bytes of memory for its data storage. So the range of locations which could be used is given by 256*64 = 16384. In other words Sprite data can only be stored in the first 16K of memory.

Within this range of memory locations (0 to 16383 decimal) there are many locations which cannot be used for Sprite data storage. In fact, there are only two available areas which *can* be used, these are:–

1. The Cassette I/0 buffer. (828 decimal to 1023 decimal.) This is used by the cassette recorder during LOAD, SAVE, PRINT# and INPUT# operations. Provided that the cassette recorder is not being used, you can fit the data for three Sprites' in here.

2. The BASIC text area. (2048 decimal to 40959 decimal.) This is, of course, fully used by the BASIC program. It is possible however, to move the start of the BASIC text area up a few bytes to release some space for Sprite data. This can be done by typing the following lines in direct mode. (You can't RUN this as a program because as soon as you start to move the bottom of BASIC you will loose the program!!)

```
NEW
POKE 642,10 : POKE 44,10 : POKE 46,10 : POKE 48,10 :
POKE 50,10 : POKE 2560,0
NEW
```

This will move the bottom of BASIC from 2048 to 2560, thus releasing 512 bytes which can be safely used to store the data for eight Sprites. (Pressing RUN/STOP and RESTORE will restore BASIC to normal.)

(Chapter 10 contains a machine code routine which moves the bottom of BASIC up to address 16384 to make room for Sprite data and the high resolution screen.)

Wherever you decide to store the Sprite data, the easiest way to get it into memory is to include it in the program in

```
100 FOR C=0 TO 63 : REM SET UP FOR
    NEXT LOOP
110 READ V : REM GET NEXT VALUE TO
    BE POKED
120 POKE 832+C,V : REM POKE THE
    VALUE INTO THE NEXT LOCATION
130 NEXT C : REM REPEAT FOR NEXT
    VALUE
140 END
150 REM THIS IS THE SPRITE DATA
160 REM IT DRAWS A LUNAR LANDER
170 DATA 0,4,0,0,8,0,0,24
180 DATA 0,0,126,0,0,255,0,1
190 DATA 255,128,1,255,128,1,255,128
200 DATA 1,231,128,0,231,0,7,255
210 DATA 224,11,255,208,23,255,232,
    39
220 DATA 255,228,75,255,210,81,255,
    138
230 DATA 96,24,6,64,24,2,64,24
240 DATA 2,224,60,7,0,0,0,0
```

DATA statements and use a FOR/NEXT loop to POKE the values in. For example the listing on page 43 stores the data for a Sprite, starting at location 832 (in the cassette buffer). 832 is 13*64, so the appropriate Sprite Data Pointer must be POKE'd with the value 13 to point to this data.

Turning Them On

No, this has nothing to do with erogenous zones (whatever they are!), it's about displaying Sprites on the screen.

A Sprite is always in one of two display states, it is either being displayed or it is not. These two states can be represented by a 1 for *ON* and a 0 for *OFF*, which means that all eight Sprites can be controlled by one register. This register is called the Sprite Enable register and it is located at address 53269 decimal. Each bit within this register controls one Sprite. Bit 0 controls Sprite 0, bit 1 controls Sprite 1, and so on.

For example to turn Sprite 4 on (without disturbing the states of other Sprites) you should use the following POKE: POKE 53269, PEEK (53269) OR 16 (see Chapter 3 for an explanation).

Colouring Sprites

Sprites can be displayed in any of the 16 colours available and there are two modes of operation, depending on the number of colours in the Sprite. These modes are:–

1. Standard colour mode

In this mode the Sprite can only be in one colour on a transparent background.

2. Multi-coloured mode

In this mode the Sprite can be in three different colours on a transparent background.

(By transparent background we mean that any dots which are not lit on the Sprite will allow whatever is underneath on the main screen to show through.)

In the standard colour mode the Sprite colour is selected by POKEing the appropriate colour value (0 to 15) into the appropriate Sprite Colour register. These registers are located at addresses 53287 decimal to 53294 decimal. There is one register for each Sprite, register 53287 is for Sprite 0, register 53288 is for Sprite 1, and so on. In each of the Sprite Colour registers only the low four bits are effective (4 bits can take 16 states and there are only 16 available colours). Figure 6.3 shows the 16 colour values which can be used.

Colour value (decimal)	*Colour name*
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light Red
11	Light Grey
12	Medium Grey
13	Light Green
14	Light Blue
15	Dark Grey

Fig. 6.3

For example, to display Sprite number 5 in Orange you should use the following POKE:–

```
POKE 53292,8
```

In multi-colour mode you sacrifice horizontal resolution

for increased colours. Instead of treating every dot independently as either coloured or transparent, the horizontal dots are read in pairs so that each pair of dots can be in one of three colours or transparent. (Two dots can have four states.) Figure 6.4 shows how the dot pairs are coded into colours.

Dot pair	*Colour taken from*
00	Transparent
01	Sprite Multicolour register 0
10	Normal Sprite colour register
11	Sprite Multicolour register 1

Fig. 6.4

The two Sprite Multicolour registers hold the values of the two extra colours. These colours are the same for all Sprites in Multicolour mode. Figure 6.5 shows where the Sprite Multicolour registers are located.

Register name	*Address (decimal)*
Sprite Multicolour 0	53285
Sprite Multicolour 1	53286

Fig. 6.5

To select multicolour mode for a Sprite you must set the appropriate bit in the Sprite Multicolour register. This is located at address 53276 decimal and, as before, bit 0 controls Sprite 0, bit 1 controls Sprite 1, and so on. If the appropriate bit is set to 1 then that Sprite will be displayed in multicolour mode.

For example, to display Sprite 2 in multicolour mode using a combination of green, light red, brown and a transparent background, you should use the following POKE's:–

```
100 REM SET MULTI-COLOUR MODE FOR
    SPRITE TWO
110 POKE 53276,PEEK(53276) OR 4
120 REM SET THE SPRITE MULTI-COLOUR
    REGISTERS
130 REM TO GREEN (5) AND LIGHT RED
    (10)
140 POKE 53285,5 : POKE 53286,10
150 REM SET THE SPRITE COLOUR
    REGISTER TO BROWN (9)
160 POKE 53289,9
```

Moving Sprites Around

We said at the beginning of this chapter that one of the main advantages of Sprites was the ease with which they could be moved around the screen. This has been achieved on the Commodore 64 by allocating a pair of registers, called the Sprite Position registers, for each Sprite. These registers hold the current position of the top left hand corner dot of the Sprite on the 320 by 200 dots high resolution screen.

The VIC chip, which is constantly redrawing the display, looks at the Sprite Position registers to find out where the Sprites should be. POKEing new values into one of these register pairs will cause the VIC chip to move the Sprite from its old position to the new one. Figure 6.6 shows the location of the Sprite position registers.

As we said earlier, the Sprite Position registers hold the location of each Sprite on the high resolution screen. However, for Sprite positioning purposes, the high resolution screen is not numbered in the way you might expect. Instead of being numbered from 0 to 319 in the X direction and 0 to 199 in the Y direction, the screen is actually numbered from 24 to 344 in the X direction and from 50 to 250 in the Y direction. The reason for this is that it allows Sprites to be smoothly moved on and off the screen without needing negative position coordinates (which the system cannot deal with).

Sprite number	*X register*	*Y register*
0	53248	53249
1	53250	53251
2	53252	53253
3	53254	53255
4	53256	53257
5	53258	53259
6	53260	53261
7	53262	53263

Fig. 6.6

For example, a Sprite positioned at 12,50 will be fully on screen in the Y direction and half on screen in the X direction. And a Sprite positioned at 0,0 will be completely off screen in both directions.

In addition, a problem occurs when using large values of X. This is because the X Sprite Position registers can only hold numbers in the range 0 to 255, and this is not enough to completely address the 320 dots wide screen. This has been overcome by using an extra register to provide each of the X Sprite Position registers with an extra high bit, thus making them all 9 bits wide. (A 9 bit register can hold numbers in the range 0 to 511.) This additional register is called the Most Significant Bit X register (MSBX for short) and it is located at address 53264 decimal. As with other bit-wise registers, bit 0 is the extra bit for Sprite 0's Sprite Position X register, bit 1 is for Sprite 1, and so on.

This does make life a little bit more difficult, because we now have to check every X coordinate to see if it is larger or smaller than 255.

If it is larger than 255 we must set the appropriate extra bit, subtract 256 from the X coordinate, and store the remainder in the appropriate Sprite Position X register.

If it is smaller than 255 we must reset the appropriate

extra bit and store the whole X coordinate in the appropriate Sprite Position X register.

The following BASIC subroutine will set up the Sprite Position registers correctly, regardless of the size of the X coordinate. Before calling this routine, the Sprite number must be placed in the SN variable, and the X,Y coordinate must be placed in the X and Y variables.

```
100 REM FIRST STORE THE Y COORDINATE
110 POKE 53249+SN*2,Y
120 REM TEST THE SIZE OF THE X
    COORDINATE
130 IF X<256 THEN 200
140 REM X>255 SO SET THE APPROPRIATE
    MSBX BIT
150 POKE 53264,PEEK(53264) OR 2↑SN
160 X=X-256
170 POKE 53248+SN*2,X : REM STORE
    THE X COORDINATE
180 RETURN
190 REM X<256 SO RESET THE
    APPROPRIATE MSBX BIT
200 POKE 53248+SN*2,X : REM STORE
    THE X COORDINATE
220 POKE 53264,PEEK(53264) AND
    NOT (2↑SN)
230 RETURN
```

(Chapter 10 contains a machine code routine which sets up the Sprite Position registers much faster.)

Bigger Sprites

Having all the Sprite screens the same size can be a bit restricting. Sometimes you may want a Sprite to be a bit higher or wider than normal to fit in with the scale of the background.

This can be accomplished, to a degree, by using the Sprite Expansion facility.

Sprites can be doubled in size, in both the X and the Y directions, by setting the appropriate bits in the X and/or Y Sprite Expansion registers. These registers are located at addresses 53277 decimal for the X Expansion register and 53271 decimal for the Y Expansion register. Within each register, bit 0 controls Sprite 0, bit 1 controls Sprite 1, and so on. If the expansion bit is a 1 then that Sprite is doubled in size in the appropriate direction.

For example, to display Sprite 3 expanded in the X direction, we must use the following POKE:–

```
POKE 53277, PEEK (53277) OR 8
```

What actually happens is that with expand on, every dot on the Sprite screen is expanded to overlay two dots on the high resolution screen. The resolution of the Sprite does not change, it is still 24 x 21 dots, but it now overlays a 48 x 42 dots area of the high resolution screen.

Getting Your Priorities Right

As we said at the beginning of this chapter, one of the most powerful facilities of Sprites is their ability to be assigned priorities. Sprite priority determines which Sprite will appear in front when two Sprites overlay each other, and whether a Sprite will appear to be in front of or behind the background.

The Sprite to Sprite priorities are built in to the system and cannot be changed. Sprite 0 has the highest priority, i.e. no other Sprite can appear to be in front of it. Sprite 1 has the next highest priority, i.e. it will pass behind Sprite 0 but in front of all the others. The other Sprite priorities follow the same pattern. This means that you need to bear the Sprite priorities in mind when deciding which Sprite to use for each character.

The Sprite to background priorities can be individually selected for each Sprite. This can be done by setting the

appropriate bit in the Sprite Background Priority register which is located at address 53275 decimal. Each bit controls one Sprite, bit 0 controls Sprite 0, bit 1 controls Sprite 1, and so on. If the appropriate bit is a 0 then that Sprite will appear to pass in front of all background graphics. If it is a 1 then that Sprite will appear to pass behind all background graphics.

For example, to make Sprite 4 pass behind the background graphics, we must use the following POKE:–

POKE 53275, PEEK (53275) OR 16

The following program illustrates the effects of Sprite priorities. It uses two Sprites, one red and one green, and a central vertical blue band of background. Try running the program and see how the Sprites and the background interact.

```
10 REM SET UP THE SPRITE DATA
15 POKE 53281,0 : POKE 53280,0
20 FOR C=0 TO 63 : READ S
25 POKE 832+C,S : NEXT C
30 POKE 2040,13 : POKE 2041,13 :
   REM SET SPRITE DATA POINTERS
40 POKE 53275,1 : REM SET THE
   BACKGROUND PRIORITIES
50 POKE 53287,5 : POKE 53288,2 :
   REM SET SPRITE COLOURS
60 PRINT "{clr/home}";CHR$(31); :
   REM CLEAR SCREEN, USE BLUE PEN
70 REM DRAW THE VERTICAL BACKGROUND
   LINE
80 FOR C=1 TO 25
82 PRINT SPC(19);CHR$(18);CHR$(32)
84 NEXT C
```

continued

```
90 POKE 53269,3 : REM ENABLE SPRITES
   0 AND 1
100 REM SET Y COORDINATE VALUES
110 POKE 53249,90 : POKE 53251,102
120 FOR X=130 TO 200 : REM X
    DIRECTION MOVEMENT LOOP
130 POKE 53248,X : POKE 53250,330-X
140 FOR D=1 TO 50 : NEXT D : REM
    DELAY LOOP
150 NEXT X
160 GOTO 120 : REM REPEAT LOOP
200 DATA 255,255,255,255,255,255,255
    ,255
210 DATA 255,255,255,255,255,255,255
    ,255
220 DATA 255,255,255,255,255,255,255
    ,255
230 DATA 255,255,255,255,255,255,255
    ,255
240 DATA 255,255,255,255,255,255,255
    ,255
250 DATA 255,255,255,255,255,255,255
    ,255
260 DATA 255,255,255,255,255,255,255
    ,255
270 DATA 255,255,255,255,255,255,255
    ,255
```

If you typed the program in correctly, you will have seen that the green Sprite (Sprite 0) passes over the red Sprite (Sprite 1), but that the green Sprite passes behind the blue background whilst the red Sprite passes in front of it!

Detecting Collisions

This is probably the most useful of the Sprite facilities, because

collision detection using standard graphics can be a lengthy and difficult process. By using Sprites however, any number of collisions can be detected simply by PEEKing a pair of registers.

Not surprisingly, it is the VIC chip which actually detects that a collision has occurred, and it reports the fact to us via two collision registers. One of these registers is used for Sprite to Sprite collisions and the other is used for Sprite to background collisions. Figure 6.7 shows the locations of these two registers.

Register name	*Address (decimal)*
Sprite to Sprite Collisions	53278
Sprite to Background Collisions	53279

Fig. 6.7

Just before we look at the way in which these registers are used, we need to define exactly what we mean by a collision.

A Sprite to Sprite collision has occurred when a non-transparent area of one Sprite overlays a non-transparent area of another Sprite.

A Sprite to background collision has occurred when a non-transparent area of a Sprite overlays any background graphics.

This means that for missile/target type applications, the collision will not occur until the missile actually hits the target, regardless of the way the missile has been drawn on the Sprite screen.

Now back to the collision registers ... In each register bit 0 refers to Sprite 0, bit 1 refers to Sprite 1, and so on. Both registers are normally set to 0, and the VIC chip sets the appropriate bits to 1 for those Sprites which have collided.

For example, if Sprite 0 has collided with Sprite 4, then the Sprite to Sprite Collision register will contain 17 decimal (bits 0 and 4 set).

Or, if Sprite 5 has collided with the background, then the Sprite to Background Collision register will contain 32 decimal (bit 5 set).

At first sight the collision registers seem quite difficult to use, because the numbers which they return don't immediately relate to individual Sprites. The problem can be easily resolved by PEEKing the registers in the right way.

For example, the following program shows the way in which the Sprite to Sprite Collision register should be PEEKed in order to discover whether Sprite 4 has collided with any other Sprites.

```
100 IF PEEK(53278) AND 16 THEN 300
110 REM CONTROL PASSES TO HERE IF
    SPRITE 4 HAS NOT COLLIDED
 .
 .
300 REM CONTROL PASSES TO HERE IF
    SPRITE 4 HAS COLLIDED
```

The AND 16 operation in line 100 masks out all bits except bit 4. If bit 4 is set then the test is true and control passes to line 300, otherwise it fails and control falls through to line 110. This technique can be used to test for any combination of Sprites on either of the collision registers.

And finally, one rather nice facility of the collision registers is that they automatically reset themselves after they have been PEEKed, so you don't need to worry about resetting them yourself!

Chapter 7

THE HIGH RESOLUTION SCREEN

The Commodore 64's high resolution screen is something of an enigma, because the User Manual tells you absolutely nothing about it! In fairness to Commodore however, this may be because the high resolution screen is not an easy facility to learn to use, the User Manual is aimed at a very basic level.

So, in this chapter we will remedy these omissions by explaining what the high resolution screen is, where it can be located and how you can use it. We have also included many short subroutines that will help you to drive the high resolution facilities.

High Resolutions?

The first thing we should take a look at is a definition for the term high resolution. Well, in the computer world the term resolution refers to the number of points on the screen which can be individually addressed.

A low resolution screen is a screen which has a small number of individually addressable points. The normal text screen is a low resolution screen because it only has 1000 individual points (40 columns by 25 rows).

Conversely, a high resolution screen is a screen which has a large number of individually addressable points. A screen is usually termed high resolution if it has more than 40000 individual points. The Commodore 64s' screen is 320 dots wide by 200 dots high giving 64000 individual points.

So that we can individually address each of these 64000 dots they are bit mapped into RAM. This means that each of the dots on the screen is controlled by a bit in memory. If the bit is a 1 then that dot will be lit, if the bit is a 0 then the dot will not be lit.

From this we can work out how much memory will be needed to store the high resolution screen. As we have seen there are 64000 dots on the screen and each byte in RAM can control 8 dots (8 bits in a byte), so we will need 8000 bytes to completely store the screen.

Locating the Screen in RAM

In the 64K of RAM available there are 8 different places where the high resolution screen could be located (65536/8000 = 8.192). However, in the standard system using BASIC, only the first two of the eight possible locations can be used. (This is because it is the Video Interface Controller chip (or VIC chip for short) which controls the high resolution screen, and in the standard system the VIC chip can only see the first 16K of RAM.)

The two locations which we can use for the high resolution screen start at addresses 0 and 8192 decimal. (The number 8192 is used because it is 2^{13} which a computer finds more convenient than the number 8000.) Unfortunately, the operating system uses most of the low memory locations, so the only available place to locate the high resolution screen is at address 8192 to 16192.

You've probably noticed that this falls right in the middle of the BASIC text area, so in order to prevent BASIC from overwriting the screen, we must move the bottom of BASIC up past address 16193 decimal. As we saw in Chapter 6 this is not too difficult, and the following lines, which must be entered in direct mode, will move the bottom of BASIC up to address 16384 decimal. (This is the closest we can get to 16193 because the bottom of BASIC can only be moved in 256 byte blocks.)

```
NEW
POKE 642,64 : POKE 44,64 : POKE 46,64 : POKE 48,64 :
POKE 50,64 : POKE 16384,0
NEW
```

(Chapter 10 contains a machine code routine which moves the bottom of BASIC for you.)

The result of moving BASIC by this amount is that we have lost 14K of program space even though the screen only takes up 8K. It is now possible however, to store the data for 32 Sprites in the space between address 2048 decimal and 4096 decimal. (It is not possible to put Sprite data between addresses 4096 decimal and 8192 decimal because the VIC chip maintains an image of the character set ROM in this area.)

As we mentioned earlier, it is the VIC chip which controls the high resolution screen, and to find out where the screen is located it looks at bit 3 of the VIC Memory Control register which is located at address 53272 decimal.

If this bit is 0 then the screen starts at address 0000, if the bit is 1 then the screen starts at address 8192 decimal. This means that when we are using BASIC we must always set this bit.

For example, to locate the high resolution scrren at address 8192 decimal POKE: POKE 53272, PEEK (53272) OR 8.

All we need to do now is to tell the VIC chip to switch to high resolution mode. This can be done by setting bit 5 of the VIC Control register which is located at address 53265 decimal. If this bit is a 1 then the screen is in high resolution mode, if it is a 0 then it is in normal text mode.

For example, to switch to high resolution mode we must use the following POKE: POKE 53265, PEEK (53265) OR 32.

(Chapter 10 contains a machine code routine which locates the high resolution screen at address 8192 and switches to high resolution mode.)

The VIC chip will now stop displaying the normal text screen and will begin displaying the high resolution screen. This introduces an immediate problem because you can no longer see what you are typing! If you are fairly careful though, you shouldn't have any problems.

The screen itself will probably be full of garbage, and this comes from two sources. Firstly, memory locations 8192 decimal to 16192 decimal are probably full of rubbish and

this will be displayed as red dots on a black background. In addition, any characters that were being displayed on the normal text screen are now displayed as different coloured dots. This happens because the normal text screen memory is now used to store the colour information for the high resolution screen (more about this later).

Obviously before we can begin using the screen we must clear out this rubbish. The coloured squares can easily be removed by printing a clear screen character (SHIFT/CLR) although they will reappear if you type anything on the keyboard. Clearing the high resolution screen can be done from BASIC by using the following routine:

```
100 FOR C=0 TO 7999
110 POKE 8192+C,0
120 NEXT C
```

(Chapter 10 contains a machine code routine which clears the screen much faster.)

Screen Organisation and Use

As we have already mentioned, the high resolution screen is 320 dots wide by 200 dots high. The obvious way to number the screen would be to have the top left corner as 0,0 and then number the X direction from 0 to 319 and the Y direction from 0 to 199. Figure 7.1 shows this logical screen numbering.

Unfortunately, the high resolution screen memory isn't organised in this way. Figure 7.2 shows the physical screen numbering.

By referring to Figures 7.1 and 7.2 you can see that the logical screen coordinates 5,0 actually refer to bit 2 of byte 0. Similarly the logical screen coordinates 10,3 actually refer to bit 5 of byte 11.

Obviously the logical numbering system is the one which we want to be using for drawing on the high resolution screen.

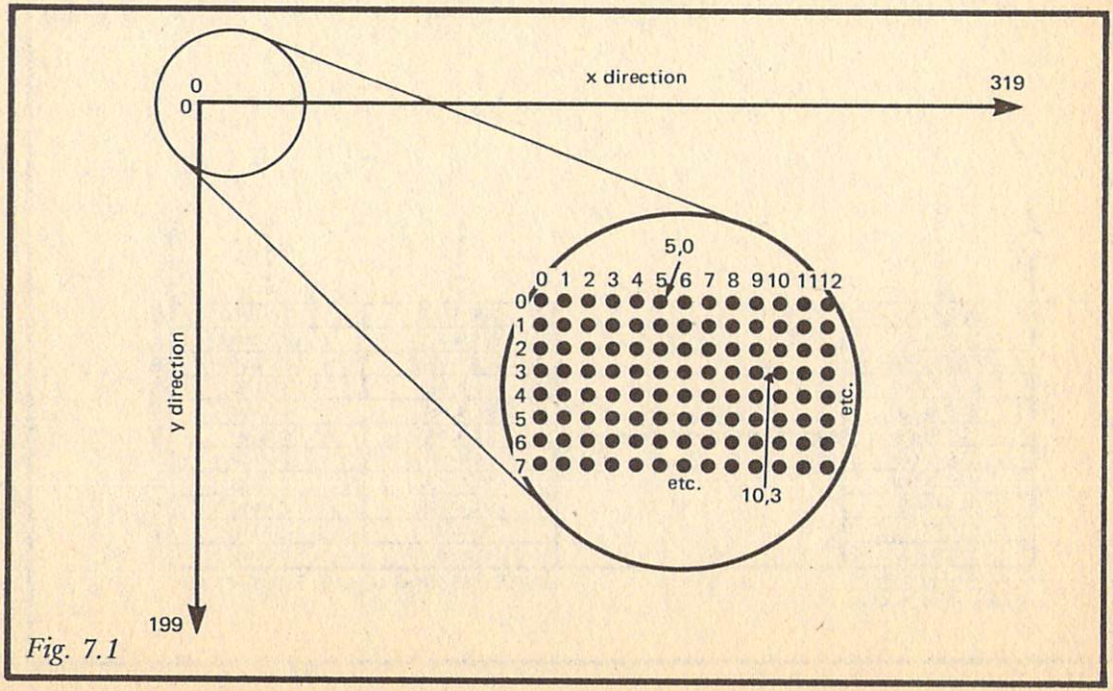

0
0
x direction
319
y direction
199
5,0
0 1 2 3 4 5 6 7 8 9 10 11 12
0
1
2
3
4
5
6
7
etc.
etc.
10,3

Fig. 7.1

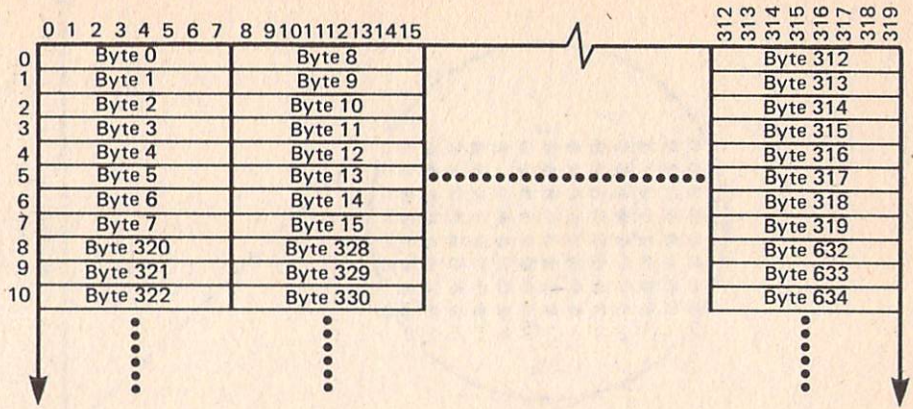

0 1 2 3 4 5 6 7 8 9101112131415
312 313 314 315 316 317 318 319
0 Byte 0 Byte 8 Byte 312
1 Byte 1 Byte 9 Byte 313
2 Byte 2 Byte 10 Byte 314
3 Byte 3 Byte 11 Byte 315
4 Byte 4 Byte 12 Byte 316
5 Byte 5 Byte 13 Byte 317
6 Byte 6 Byte 14 Byte 318
7 Byte 7 Byte 15 Byte 319
8 Byte 320 Byte 328 Byte 632
9 Byte 321 Byte 329 Byte 633
10 Byte 322 Byte 330 Byte 634

Fig. 7.2

This means that we will need a conversion routine to convert from the logical numbering system to the physical screen organisation.

The following BASIC routine will take the logical screen location in the X and Y variables, convert it to a physical screen byte and then set the appropriate bit within this byte to light the dot on the screen.

```
100 REM FIRST SORT OUT THE BYTE TO
    BE CHANGED
110 BY=8192+INT(Y/8)*320+INT(X/8)
    *8+(Y AND 7)
120 REM NOW SORT OUT AND SET THE
    BIT TO BE CHANGED
130 POKE BY,PEEK(BY) OR
    2↑(7-(X AND 7))
```

(Chapter 10 contains a machine code routine which makes plotting points much easier.)

Once you can plot a point on the screen, drawing lines becomes fairly easy. Lines are simply successive plots in an ordered sequence.

The following BASIC routine will draw a line between the point indicated by the LX and LY variables (last X and last Y) and the point indicated by the NX and NY variables. On exit from the routine the contents of LX and LY are set to the values in NX and NY. This is so that you can draw continuous objects (boxes etc.) by simply supplying new values for NX and NY.

```
100 REM FIRST SORT OUT THE DIRECTION
    IN WHICH THE LINE MUST GO
110 XS=SGN(NX-LX) : YS=SGN(NY-LY)
120 REM AND THE THE STEP VALUES
130 DX=ABS(NX-LX) : DY=ABS(NY-LY)
```

continued

```
140 EX=DY
150 REM NOW DRAW THE LINE BY
    REPEATED PLOTS
160 IF EX>0 THEN EX=EX-DX : LY=LY+YS
170 IF EX<=0 THEN EX=EX+DY :
    LX=LX+XS
180 GOSUB 1000
190 IF LX<>NX OR LY<>NY THEN 160
200 RETURN
999 REM THIS IS THE PLOT LX,LY
    ROUTINE
1000 BY=8192+INT(LY/8)*320+
     INT(LX/8)*8+(LY AND 7)
1010 POKE BY,PEEK(BY) OR
     2↑(7-(LX AND 7))
1020 RETURN
```

(Chapter 10 contains a machine code line draw routine which works much faster.)

Once you can draw lines all sorts of drawings can be constructed quite simply. For example, the following BASIC routine will draw a circle with a centre of XC and YC and a radius of RD.

```
5 REM PLOT THE FIRST POINT
10 LX=XC+RD : LY=YC
20 GOSUB 1000
25 REM NOW DRAW THE REST OF CIRCLE
30 FOR A=0 TO 360 STEP 10 : REM
   TEN DEGREE STEPS
40 NX=INT(RD*COS(A*π/180)+XC) : REM
   NEW X COORDINATE
50 NY=INT(RD*SIN(A*π/180)+YC) : REM
   NEW Y COORDINATE
60 GOSUB 100
```

```
70 NEXT A
80 END
.
.
100 REM THE ROUTINE TO DRAW A LINE
    FROM NX,NY TO LX,LY FROM HERE
.
.
1000 REM THE ROUTINE TO PLOT A POINT
     AT LX,LY GOES IN HERE
```

Using Colour

As we mentioned earlier, the colour information for the high resolution screen comes from the normal text screen area, which is located at address 1024 decimal to 2023 decimal. Each text screen location holds the colour information for an 8 by 8 dots square on the high resolution screen. The byte at address 1024 decimal holds the colour information for bytes 0 to 7 of the high resolution screen, the byte at address 1025 decimal holds the colour information for bytes 8 to 15 of the high resolution screen, and so on. (See Figure 7.2.)

Within each colour byte the four high order bits (bits 4 to 7) contain the colour for any bit on the high resolution screen which is set to a 1 and the four low order bits (bits 0 to 3) contain the colour for any bit on the high resolution screen which is reset to 0.

This means of course, that the colour memory has a much lower resolution than the screen memory, so you can only colour the screen in blocks of 64 bits.

Just as with Sprite colours, the high resolution screen can operate in two colour modes. These two modes are:–

1. Standard colour mode

In this mode any dot on the high resolution screen can only be in one of two colours. These colours are simply taken from the high and low bits of the text screen memory

as described above. Usually the whole high resolution screen will be in the same two colours, which means that the composite value of the colour can be written all through the text screen memory.

The following BASIC routine will colour the whole high resolution screen using the foreground colour in the FC variable and the background colour in the BC variable.

```
100 REM FIRST FORM THE COMPOSITE
    COLOUR VALUE
110 CC=FC*16+(BC AND 15)
120 REM NOW WRITE THIS COLOUR INTO
    THE TEXT SCREEN MEMORY
130 FOR I=0 TO 999
140 POKE 1024+I,CC
150 NEXT I
```

(Chapter 10 contains a machine code routine which sets up the screen colours much faster.)

2. Extended colour mode

In this mode you sacrifice horizontal resolution for an increased number of colours. Instead of every dot being in one of two colours, the horizontal dots are now read in pairs so that each pair of dots can be in one of four colours. Figure 7.3 shows how the dot pairs are code into colours.

Dot pair	*Colour taken from*
00	Background colour register 0
01	High 4 bits of the text screen memory
10	Low 4 bits of the text screen memory
11	Normal colour memory

Fig. 7.3

The Background colour register 0 is located at address

53281 decimal and it normally holds the text screen background colour. The colour in this register is the same for the whole of the high resolution screen, i.e. the high resolution screen background must all be the same colour.

The normal colour memory, which is located at address 55296 decimal to 56295 decimal, is where the colour for the characters on the text screen is normally held. In extended colour mode each location can be used to hold the colour for an 8 by 8 dots square on the high resolution screen.

For example, location 55296 holds the colour for any dot pairs set to 11 in bytes 0 to 7 of the high resolution screen, 55297 holds the colour for any dot pairs set to 11 in bytes 8 to 15, and so on.

To select extended colour mode for the high resolution screen you must set bit 4 of the second VIC control register which is located at address 53270 decimal. For example, to select extended colour mode for the high resolution screen you must use the following POKE: POKE 53270, PEEK (53270) OR 16.

Drawing Complex Shapes

By using the BASIC PLOT and LINEDRAW routines described above, or the much faster machine code routines described in Chapter 10, it is possible to draw almost any complex shape.

All that is required is a coordinate file which holds all the necessary drawing coordinates and status information such as skip to the next point and end of file. The easiest way to store this sort of data is in BASIC DATA statements.

We can store the X,Y coordinates as simple pairs of numbers in the range 0–319 for the X coordinate and 0–199 for the Y coordinate. For the status information we can use –1 to indicate skip to the next point and –2 to indicate the end of the file.

For example, consider the following DATA statement:

```
100 DATA–1,123,34,256,15,–1,304,112,26,132,–2
```

This means – skip to the point 123,34 and then draw to 256,15, skip to the point 304,112 and then draw to 26,132 and then end.

The following BASIC program should make this a little clearer! It draws the outline of the lunar lander we used in Chapter 6.

First move the bottom of BASIC up to 16384 decimal to make room for the screen.

```
NEW
POKE 642,64 : POKE 44,64 : POKE 46,64 : POKE 48,64 :
POKE 50,64 : POKE 16384,0
NEW
```

Now enter and run the program:–

```
10 REM THIS IS THE DATA FOR THE
   DRAWING
20 DATA -1,90,30,90,100,140,100,140
   ,30,90,30
30 DATA -1,60,100,50,110,50,140,60
   ,150,170,150
40 DATA 180,140,180,110,170,100,60
   ,100,-1,90
50 DATA 40,70,50,70,80,90,100,-1
   ,140,40,160,50
60 DATA 160,80,140,100,-1,60,100,50
   ,100,10,140
70 DATA 10,190,-1,0,190,20,190,-1
   ,50,130,10
80 DATA 170,-1,170,100,180,100,220
   ,140,220,190
90 DATA -1,210,190,230,190,-1,180
   ,130,220,170
100 DATA -1,100,190,130,190,-1,110
    ,190,110,80
```

```
110 DATA 120,80,120,190,-1,120,30
    ,120,10,-1
120 DATA 110,20,130,0,-2
200 REM ENTER HIGH RESOLUTION MODE
210 POKE 53265,PEEK(53265) OR 32
220 REM LOCATE THE HIGH RES SCREEN
230 POKE 53272,PEEK(53272) OR 8
240 REM CLEAR HIGH RES SCREEN
250 FOR I=0 TO 7999 : POKE 8192+I,0
    : NEXT I
260 REM SET BLUE BACKGROUND AND
    WHITE FOREGROUND
270 FOR I=0 TO 999 : POKE 1024+I,22
    : NEXT I
280 REM THIS IS THE DRAWING LOOP
300 READ X
310 REM TEST FOR STATUS INFO
320 IF X>=0 THEN 600 : REM DRAW
330 IF X=-1 THEN 400 : REM SKIP
340 IF X=-2 THEN PRINT "{home}" :
    REM 'PARK' THE TEXT CURSOR
350 GOTO 350 : REM ENLESS LOOP TO
    FINISH
380 REM THIS IS THE PLOT ROUTINE
390 REM GET X & Y COORDINATES
400 READ X : READ Y : LX=X : LY=Y
410 GOSUB 800
420 GOTO 300
580 REM THIS IS THE DRAW ROUTINE
590 REM GET Y COORDINATE
600 READ Y
610 XS=SGN(X-LX) : YS=SGN(Y-LY)
```

continued

```
620 DX=ABS(X-LX) : DY=ABS(Y-LY)
630 EX=DY
640 GOSUB 800
650 IF EX>0 THEN EX=EX-DX : LY=LY+YS
660 IF EX<=0 THEN EX=EX+DY :
    LX=LX+XS
670 GOSUB 800
680 IF LX<>X OR LY<>Y THEN 650
690 GOTO 300
790 REM THIS IS THE PLOT ROUTINE
800 BY=8192+INT(LY/8)*320+
    INT(LX/8)*8+(LY AND 7)
810 POKE BY,PEEK(BY) OR 2↑(7-(LX
    AND 7))
820 RETURN
```

This program takes a very long time to run (we timed it at 3 minutes 45 seconds). This is mainly because CBM 64 BASIC is very slow and the program uses a couple of long FOR/NEXT loops which take up most of the time. Using the machine code routines in Chapter 10 however, the same program takes only 2.5 seconds to run!

Chapter 8

SOUND

Computer generated sound and music is nothing new, every time you turn on a radio you can hear some excellent examples of it. Not until fairly recently however, have home computers been able to generate high quality sound with any degree of control. The sound generator chip fitted to the Commodore 64 is one of the most flexible devices currently available, and it can be used to produce a wide variety of sounds. From a programmers point of view however, this extra flexibility can mean extra complexity, because there are a large number of parameters which must be specified to produce even the simplest of sounds.

In this chapter we are going to try and simplify these complex parameters by describing exactly what they are for and how they are used. What we will not try to do however, is to present lots of little tunes and sounds for you to type in. We have chosen this approach because sound is an extremely flexible medium and its uses can only be discovered by experimentation.

Simple Sound

Sounds are generated by vibrating objects. These vibrations are transmitted to the air around the objects and eventually reach our ears where our brains decode the vibrations into sound images. From our experience we know that there is an enormous variety of sounds, ranging from the song of a bird to the rumble of thunder.

For our purposes we can define a number of variables which combine to give each sound its individuality. These are:

1. Pitch

Pitch depends upon the frequency of the vibrations. A high note vibrates rapidly and has a high frequency. A low note vibrates slowly and has a low frequency.

2. Timbre

Timbre is the quality of a sound. For example, a vibrating violin string and a vibrating tuning fork sound different, even though they may be vibrating at the same frequency. This occurs because objects vibrate at many frequencies, all at the same time. One of these frequencies, called the fundamental, is the dominant one and gives the sound its pitch. The other frequencies, called overtones or harmonics, are all integer multiples of the fundamental. The harmonics interfere with each other, and with the fundamental, to produce an overall sound. The many different sound qualities result from the fact that different vibrating objects produce different harmonics and thus different overall sounds.

3. The shape of the sound

Any sound goes through a number of stages in its life, from first being produced to finally dying away. Some sounds, like a gun shot, grow very rapidly to some peak volume and die away just as quickly. Other sounds, like a violin string being bowed, grow fairly quickly to some peak volume but take a long time to die away. The shape of a sound is often called its envelope and it consists of four distinct stages.

The *Attack* rate. This is the rate at which the sound rises to maximum volume.

The *Decay* rate. This is the rate at which the sound falls from its maximum volume to its average volume.

The *Sustain* level. This is the average volume which the sound has.

The *Release* rate. This is the rate at which the sound dies away from its sustain level to zero.

On the Commodore 64 we have control over all of these variables. This allows us to produce a wide variety of sounds, ranging from music to sound effects.

An Outline of the Commodore Sound System

The Commodore 64 sound system is controlled by the Sound Interface Device, or SID chip for short. Internally it contains three completely separate sound sources called voices. Figure 8.1 shows how the three voices are organised.

Each of the voices can generate four different waveforms over the frequency range 0 to 4000 Hz. Each waveform has a different wave shape and produces different harmonics. The four waveforms are, Triangular, Sawtooth, Rectangular and White noise. Figure 8.2 shows the general shape of each of these waveforms.

The envelope shapers control the shape of the sound by applying user specified values for the sustain level and the attack, decay and release rates (called the ADSR envelope). Figure 8.3. shows a typical ADSR envelope.

The outputs of the three envelope shapers are then brought together and fed into the tone filtering stage. Filtering is used to remove unwanted frequencies from the sound (mainly used for specific sound effects). There are three types of filter available, a High pass filter which rejects frequencies below the cutoff, a Low pass filter which rejects frequencies above the cutoff and a Band pass filter which rejects frequencies both above and below the cutoff. Figure 8.4. shows the effect of these three types of tone filter.

Now let's go on and look at the different parts of the sound system in more detail.

The Waveform Generators

The three waveform generators have two main functions, these are:–

1. To produce a sound of the selected frequency in the selected waveform.

2. To allow the sound output to be turned on and off.

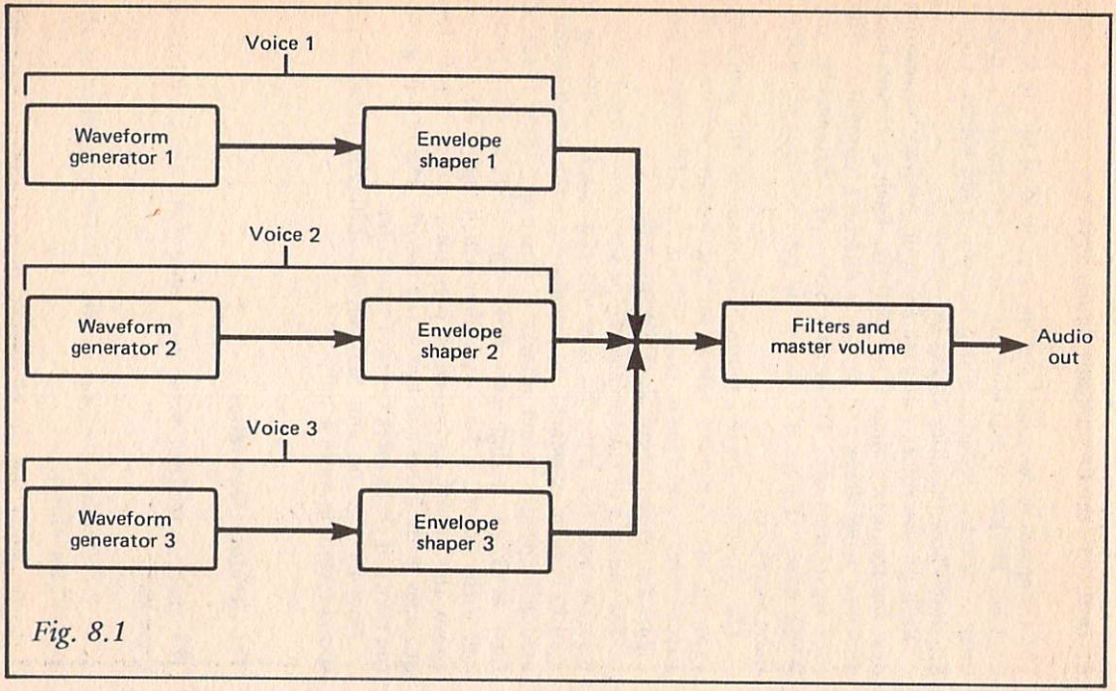

Voice 1
Waveform generator 1
Envelope shaper 1
Voice 2
Waveform generator 2
Envelope shaper 2
Voice 3
Waveform generator 3
Envelope shaper 3
Filters and master volume
Audio out

Fig. 8.1

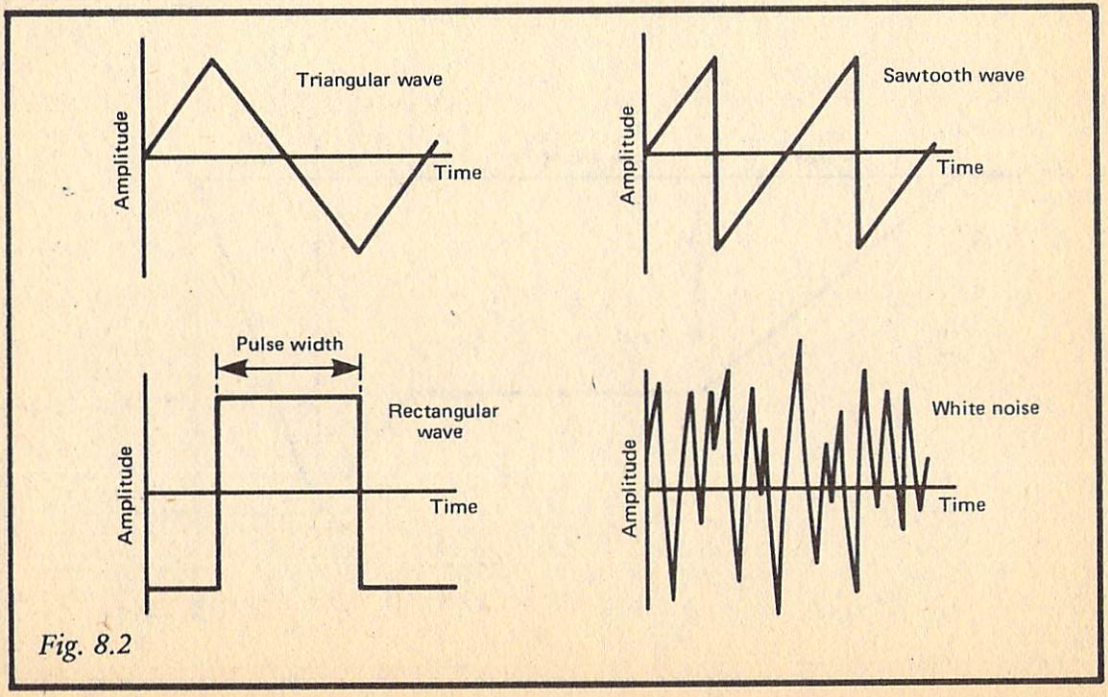

Triangular wave
Amplitude
Time
Sawtooth wave
Amplitude
Time
Pulse width
Rectangular wave
Amplitude
Time
White noise
Amplitude
Time

Fig. 8.2

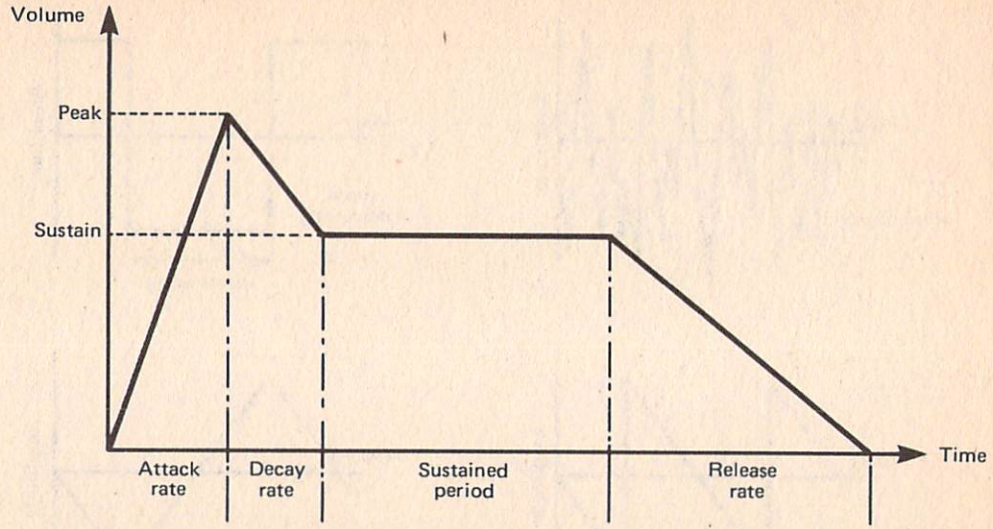
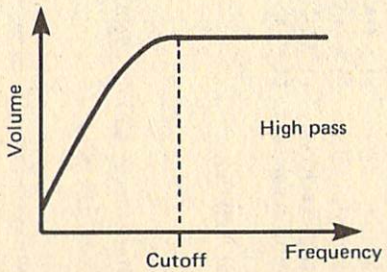
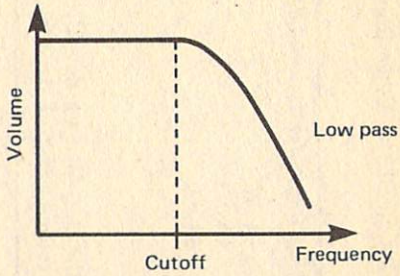

Volume
Peak
Sustain
Attack rate
Decay rate
Sustained period
Release rate
Time

Fig. 8.3


Volume
High pass
Cutoff
Frequency


Volume
Low pass
Cutoff
Frequency

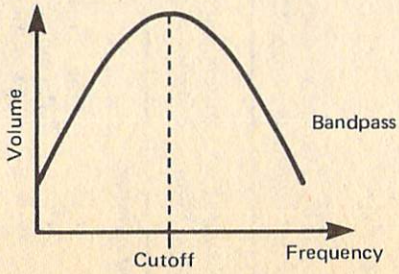

Volume
Bandpass
Cutoff
Frequency

Fig. 8.4

For each waveform generator, both of these functions are performed by one register, called the Voice Control register. Figure 8.5 shows the location of each of the Voice Control registers.

Voice number	*Voice Control register address (decimal)*
1	54276
2	54283 *
3	54290

Fig. 8.5

*NOTE – In some versions of the User Manual the address of this register is incorrectly specified as 54288 decimal.

Within these registers, each bit is used to control a separate facility. Figure 8.6 shows what each bit is used for.

Bit number	*Function*	
7	White noise select	(1 = ON)
6	Rectangular waveform select	(1 = ON)
5	Sawtooth waveform select	(1 = ON)
4	Triangular waveform select	(1 = ON)
3		
2	Used for special facilities	
1		
0	Gate bit (1 = Start attack/decay/sustain cycle) (0 = Start Release cycle)	

Figure 8.6

It is the Gate bit which allows us to turn the output on and off. Since it is the least significant bit in the register it either adds or subtracts 1 from the decimal value of the waveform,

depending on whether it is set or not. Figure 8.7 shows the decimal values to be POKE'd into the Voice Control registers both with and without the gate bit set.

Waveform type	*Attack/decay/sustain Gate bit = 1*	*Release Gate bit = 0*
White noise	129	128
Rectangular	65	64
Sawtooth	33	32
Triangular	17	16

Fig. 8.7

The following BASIC routine shows a general procedure for producing a sound. The program assumes that the Release value of the waveform (128, 64, 32 or 16) is stored in the variable WF and the Voice number to be used (1, 2 or 3) is stored in the variable VN.

```
100 REM CALCULATE THE VOICE CONTROL
    REGISTER ADDRESS
110 VC=54276+7*(VN-1)
120 REM START ATTACK/DECAY/SUSTAIN
    CYCLE
130 POKE VC,WF+1
140 REM NOW WAIT FOR THE LENGTH OF
    THE SOUND
150 FOR C=1 TO SL : NEXT C : REM
    SL IS THE SOUND LENGTH
160 REM NOW START THE RELEASE CYCLE
170 POKE VC,WF
```

The sound length, in variable SL, determines how long the sound will continue at its sustain level. If you are generating a musical note then this length will have a specific value depending on the type of note. The basic musical note is the

the crotchet which lasts for one beat, all other note lengths are multiples of this length. Figure 8.8 shows the relationship between some of the common notes.

Note name	*Note length (in crotchets)*
Semiquaver	0.25
Quaver	0.5
Crotchet	1
Minim	2
Semibreve	4

Fig. 8.8

For any piece of music you will need to determine by trial and error how long to make the FOR/NEXT loop to get a crotchet to sound the correct length. Once you have found this value though, all the others can be easily calculated from it.

Perfect Pitch?

Selecting the correct frequency for a sound is obviously much more important for music than is for sound effects. Fortunately, the system allows very precise control over the frequency so that musical notes can be accurately pitched. This fine control is achieved by using a 16 bit value to specify the frequency value. Naturally, two registers are needed to hold this 16 bit value, these are the High Frequency Control register and the Low Frequency Control register. Figure 8.9 shows where these registers are located for each of the three voices.

Each of the waveform generators can output frequencies in the range 0 to 4000 Hz. Within this range of frequencies lies a full 8 octaves of the musical scale. (About the same as a piano, which has a range of 7.5 octaves.) Appendix M in the User Manual gives a complete list of the high and low frequency values which must be POKE'd into the Frequency Control registers to produce the standard musical scale.

Voice number	*High frequency control register (decimal)*	*Low frequency control register (decimal)*
1	54273	54272
2	54280	54279
3	54287	54286

Fig. 8.9

At this point we ought to take a look at the differences between the real musical scale and a computer generated scale. The real musical scale is a fascinating mathematical series, with simple ratios between successive notes. The root note of this scale is usually taken to be A above middle C which has a frequency of exactly 440 Hz. (This is the A in octave 4 of appendix M in the User Manual.) The other notes in this scale are separated by the simple ratios in the mathematical series mentioned above. Computers on the other hand, find it much easier to use a more regular separation between notes, and they arrange that each note is $^{12}\sqrt{2}$ (the twelfth root of two) times the previous one, with A above middle C fixed at 440 Hz. This gives a very close approximation of the real musical scale and you need a very good ear to be able to spot the differences. Figure 8.10 shows the difference between A above middle C and middle C for the real scale and the note values in appendix M of the User Manual.

Note name	*Commodore frequency*	*Real note frequency*
A above middle C	440.0 Hz	440.0 Hz
Middle C	261.6 Hz	264.0 Hz

Fig. 8.10

The rectangular waveform also requires an additional parameter to specify the pulse width. The pulse width is a

measure of how wide the high part of the rectangular wave will be. (See Figure 8.2.)

For each voice the pulse width is specified as a 12 bit number, which gives 4096 possible values. The system is set up so that a pulse width of 4095 gives a rectangular wave which is permanently high, and a pulse width of 0 gives a rectangular wave which is permenantly low. Ratios in between these give varying lengths of the high part of the wave, 2048 for example, gives a true square wave.

The 12 bit pulse widths are specified using two registers, the High Pulse Width register and the Low Pulse Width register. Figure 8.11 shows where these registers are located for each of the voices.

Voice number	*High Pulse Width register (decimal)*	*Low Pulse Width register (decimal)*
1	54275	54274
2	54282	54281
3	54289	54288

Fig. 8.11

The high four bits of the pulse width are specified in the High Pulse Width register and the low eight bits are specified in the Low Pulse Width register.

The Envelope Shape

As we have seen, the ADSR envelope can be individually specified for each of the three voices. Conveniently, the ADSR variables each have 16 different steps, which means that only two registers are needed to store them all. (4 bits can take 16 different states). So the complete ADSR envelope is specified in the Attack/decay Cycle Control register and the Sustain/release Cycle Control register. Figure 8.12 shows the location of these registers for each of the voices.

Voice number	*Attack/decay Control register (decimal)*	*Sustain/release Control register (decimal)*
1	54277	54278
2	54284	54285
3	54291	54292

Fig. 8.12

Within the Attack/decay registers, the high 4 bits specify the attack rate and the low 4 bits specify the decay rate. Similarly, within the Sustain/release registers, the high 4 bits control the sustain level and the low 4 bits control the release rate. Figure 8.13 shows the levels and rates which can be selected for each of the ADSR variables.

Register value	*Attack rate*	*Decay/release rate*	*Sustain level*
0	2 mS	6 mS	0%
1	8 mS	24 mS	6%
2	16 mS	48 mS	13%
3	24 mS	72 mS	20%
4	38 mS	114 mS	26%
5	56 mS	168 mS	33%
6	68 mS	204 mS	40%
7	80 mS	240 mS	46%
8	100 mS	300 mS	53%
9	250 mS	750 mS	60%
10	500 mS	1.5 S	66%
11	800 mS	2.4 S	73%
12	1 S	3 S	80%
13	3 S	9 S	86%
14	5 S	15 S	93%
15	8 S	24 S	100%

Fig. 8.13

Tone Filtering

Using the filters is largely a matter of trial and error. There are no right values to use for any particular sound. In any case, as we said earlier, filtering probably has its main application in the sound effects field. Certainly, we have not found it necessary to use them at all when playing music, but then we're tone deaf!

The three types of filter which we described earlier are actually all part of one very clever programmable filter within the SID chip. The filter can be programmed to filter any combination of the three voices, using any combination of the three types of filter. The filtering is done by using two control registers. One, called the Voice Input Control register, selects the voices to be filtered. It is located at address 54295 decimal. Figure 8.14 shows how the bits within this register are allocated.

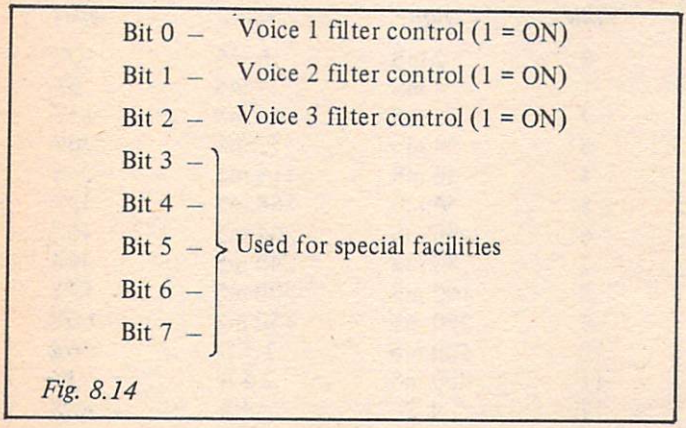


Fig. 8.14

The other register, called the Filter Mode Control register, selects the filters to be used. This register, which is located at address 54296, is also the Master Volume Control register for the whole sound system. Figure 8.15 shows how the bits within this register are allocated.

Bit 0 –	Master volume control
Bit 1 –	
Bit 2 –	
Bit 3 –	
Bit 4 –	Select low pass filter (1 = ON)
Bit 5 –	Select band pass filter (1 = ON)
Bit 6 –	Select high pass filter (1 = ON)
Bit 7 –	Used for special facilities

Fig. 8.15

Once you have decided which voices to filter and which filters to use, you must set up the cutoff frequencies for each filter you are using.

The cutoff frequency is the frequency at which the filter starts to take effect. For example, at the cutoff frequency the high pass filter will begin rejecting. As the input frequency falls the rejection will increase until the input is almost totally rejected. The range over which each of the filters can operate is 30 to 12000 Hz.

The cutoff frequencies are specified as an 11 bit number in two registers. These are, the High Frequency Cutoff register, located at address 54294 decimal, and the Low Frequency Cutoff register, located at address 54293 decimal. The high 8 bits of the cutoff frequency are specified in the High Frequency Cutoff register and the low 3 bits are specified in bits 0–2 of the Low Frequency Cutoff register. (No, we don't know why it is done in this strange way either!)

The following program illustrates the way the filters are programmed. It assumes that the voice to be filtered is in variable FV, the filter type is in variable FT (1 for low pass, 2 for band pass and 3 for high pass), the cutoff frequency is in variable CF and the master volume setting is in variable MV (0 is the quietest and 15 is the loudest).

```
100 REM SET UP THE INPUT CONTROL
    REGISTER
120 POKE 54295,PEEK(54295) OR 2↑FV
130 REM SET UP THE FILTER TYPE
140 POKE 54296,PEEK(54296) OR 8*
    (2↑FT)
150 REM AND THE MASTER VOLUME
160 POKE 54296,PEEK(54296) OR
    (MV AND 15)
170 REM NOW SET UP THE CUT OFF
    FREQUENCY
180 POKE 54293,CF AND 7 : REM SET
    LOW BITS
190 POKE 54294,(CF-(CF AND 7))/8 :
    REM SET HIGH BITS
```

Making Music

We said earlier that Appendix M in the User Manual lists the high and low frequency values for 8 octaves of the musical scale. Obviously, when you are converting a piece of music to run on the Commodore 64, you don't want to have to waste space storing all of the frequency values. What you really want is some way of calculating them all.

The simplest way of calculating the frequencies is to make use of the fact that the notes in one octave are twice the frequency of the notes in the octave below. This means that if we store the frequencies for the highest octave we can calculate all the others by successively dividing by 2.

To do this easily the high and low frequency values must be combined into one 16 bit number. This can be done as follows:–

16 bit freq = high freq * 256 + low freq

In order to recover the high and low frequency values from

a 16 bit number, we must be able to convert back again! This can be done as follows:–

high freq = INT (16 bit freq/256)
low freq = 16 bit freq – high freq

If you calculate the 16 bit frequency values for the 12 notes in the highest octave of Appendix M you get the values given in Figure 8.16.

Note name	*High frequency*	*Low frequency*	*16 bit frequency*
C	137	43	35115
C#	145	83	37203
D	153	247	39415
D#	163	31	41759
E	172	210	44242
F	183	25	46873
F#	193	252	49660
G	205	133	52613
G#	217	189	55741
A	230	176	59056
A#	244	103	62567
B	258	241	66289*

Fig. 8.16

*NOTE – This value is not included in the list in Appendix M because it is greater than 65535 and cannot be played by the SID chip. We have included it so that you can use it to calculate the B note frequencies in the lower octaves.

The following routine shows how the values for the highest octave can be used to calculate all the others. The routine assumes that the 12 note values for the highest octave are stored in the H0(12) array, the required note name is stored

in NN$ and the required octave is stored in OC. It also needs the note names to be stored in the N$(12) array. The routine will return with the high frequency value for the required note in HF and the low frequency value in LF.

```
100 REM FIRST FIND THE NOTE NAME
110 X=0 : REM INITIALIZE POINTER
120 FOR N=1 TO 12 : IF NN$=N$(N)
    THEN X=N
130 NEXT N : IF X=0 THEN PRINT
    "NOTE NOT FOUND" : END
140 REM NOTE POSITION NOW IN X
150 REM GET THE HIGHEST OCTAVE
    VALUE FOR THIS NOTE
160 NV=HO(X)
170 REM DO WE WANT THE HIGHEST
    OCTAVE?
180 IF OC=7 THEN 240 : REM JUMP
    IF WE DO
190 REM ELSE START DIVIDING BY 2
200 FOR C=6 TO OC STEP -1
210 NV=NV/2 : REM DIVIDE NOTE
    VALUE BY 2
220 NEXT C
230 REM NOW RECVER HIGH AND LOW
    FREQUENCY VALUES
240 HF=INT(NV/256) : LF=NV-HF*256
```

Sound Effects

There are a couple of additional facilities in the SID chip which are very useful when generating sound effects. (Although they can also be used in music.)

The first of these is Dynamic Frequency Variation. This is achieved by making the output of one of the waveform generators dependant upon the output of one of the others.

To make this as simple as possible, waveform generator 3 has a special digital output which produces a digitised version of the wave being generated.

For example, if a sawtooth wave is being generated then the digital output varies from 0 to 255 decimal in sympathy with the sawtooth wave. This value can then be used to influence the frequency value for one of the other generators.

This digital output is only available from waveform generator 3 and it is located at address 54299 decimal. In addition, should you not want the output of generator 3 to be audible whilst you are using its digital output, bit 7 of the Filter Mode Control register will cut off generator 3's output when it is set to 1. (See Figure 8.15.)

The next special facility is called Synchronisation and it involves the combination of the outputs of two waveform generators using the logical AND function. This can be used to provide a whole range of very interesting sounds, particularly if one of the voices is generating white noise.

The synchronisation facility is available on all the voices and it is controlled by bit 1 of the Voice Control register. Figure 8.17 shows which waveform generators are synchronised by each Voice Control register.

Voice Control register	*Address (decimal)*	*Waveform generators synchronised*
1	54276	1 and 3
2	54283	2 and 1
3	54290	3 and 2

Fig. 8.17

The last special facility is called Ring Modulation and it involves combining the output of two voices in a special and complex way. The result of this combination produces hollow bell-like sounds, which are particularly useful for producing clock chimes or any bell sound.

Ring modulation is available on all voices and, like synchronisation, it is controlled by the Voice Control registers. Figure 8.18 shows which waveform generators are ring modulated by each Voice Control register.

Voice Control register	*Address (decimal)*	*Waveform generators ring modulated*
1	54276	1 and 3
2	54283	2 and 1
3	54290	3 and 2

Fig. 8.18

As we said at the beginning of this chapter, the only way to discover the limits of the SID chips facilities is by experimentation. All we have been able to do in this chapter is to give you a few pointers in the right directions, it is now up to you to discover just what computer sound can do for you.

Chapter 9

MACHINE CODE

Teaching you to program in machine code is totally beyond the scope of this book. Indeed, this topic on its own could easily fill several books. However, if you read computing magazines, then from time to time you will most likely find several small programs or parts of programs for the Commodore 64 that are written in machine code. And you may think that it would be nice to be able to enter them and get them running. To this end we will take a look now at how to enter machine code routines that are presented as a HEX dump and how to access them from BASIC. Then in the next chapter we will present a fairly large HEX dump which, if you enter it, will add several extra and very useful new keywords to the BASIC interpreter.

SYS and USR

There are several ways to use machine code routines from BASIC on the Commodore 64, the two most useful of these being to use the keywords SYS or USR.

The BASIC statement SYS xxxxx causes the BASIC interpreter to execute a call to a machine code subroutine at address xxxxx where xxxxx is a decimal number in the range 0 – 65535. To return to BASIC, the machine code subroutine must end with a return from subroutine instruction (RTS). If the SYS statement is encountered in a BASIC program then control will be passed back to the next statement in the program after the SYS statement. If the SYS statement is encountered in direct mode then control will be passed to direct mode.

If machine code subroutines need to have parameters passed to them and returned by them then this can be achieved using the BASIC PEEK and POKE facilities to place

data in known locations in memory that the machine code can access using the machine code equivalents to PEEK and POKE.

The SYS statement is the easiest way of accessing machine code routines from BASIC as there can be any number of SYS statements in a program each of which contains the address of the routine it is to call. This means that many separate routines can be accessed from within a single program with ease.

The other common way to access machine code routines from BASIC is to use the BASIC function USR(X). The USR function is more difficult to use than the SYS system but it does have advantages under certain circumstances. The main advantage of USR over SYS is that USR allows a single parameter to be passed to the machine code subroutine and it allows the routine to pass a single parameter back as an answer. It is the fact that USR returns a value that makes it a function. i.e. It has to be used under circumstances like:–

```
10 LET X=USR(0)
```

Where the value returned by USR would be assigned to the variable X and the value passed to the subroutine would be the value of the expression in the brackets. (Here zero.)

By this stage you might be wondering about how the BASIC program knows where to go to find the machine code subroutine when using USR. Well, the answer is that the address of the subroutine has to be POKEd into two special locations in low memory. These are locations 785 and 786 (decimal). It is this need to POKE the subroutine address into memory when using USR that makes it generally less convenient to use than SYS.

Two locations (here 785 and 786) are generally required to store an address in memory as an address can be any number in the range 0 – 65535 (0 – FFFF Hex) and a single memory location can only hold a number in the range 0 – 255(0 – FF Hex).

While we are on the subject of the Hex and decimal representations of numbers, it may be a good time to take a

closer look at Hex numbers in general and at their conversion to and from decimal.

Hex to Decimal and Back

The way we are taught to count at school is based on the number 10. This means that all of our numbers can be written down using only 10 different symbols (0,1,2,3,4,5,6, 7,8 and 9). When we need to write down a number greater than 9, we use a digit's position within the number to increase the values of our 10 symbols. For example, in the number 36 the position within the number of the 3 gives it the value thirty and not its face value of three. Each time a digit is moved one position to the left, its value increases by a factor of 10. This means that the rightmost position in a whole number contains the units digit. The next position contains the tens, the next contains the hundreds, then the thousands and so on for as many digits as are needed to represent the required value. This system works well and it is the one that we have all become used to from a very early age. But the computer, which is made of thousands of on/off switches, has difficulty in coping with so many different symbols. A simple on/off switch only has, by its very nature, two different states (on and off), and it is, therefore, very much easier to represent numbers within a computer using a system based on the number 2, i.e. using the two different symbols 0 and 1. As with our usual numbering system, position is used when we need to represent a number that is bigger than can be achieved with a single digit. With the 10 based system, a digit's value increases by a factor of 10 with a one position shift to the left. With the 2 based system, a one position shift to the left multiplies a digit by a factor of two. This means that the rightmost position in a whole number to the base 2 (a binary number) still contains the units digit. But now the next position to the left contains the twos, the next position contains the fours, then the eights and so on, each position to the left being twice the current position. The number 36 used in the last example when converted to binary becomes

100100 which equals 36 by the following reasoning: 100100 is one thirty two, no sixteens, no eights, one four, no twos and no units. Adding together the values of the positions containing the ones gives 32 + 4 which is 36.

This system is fine for computers, which manage to perform calculations with it at fantastic speeds, but it is much more difficult for us than our usual decimal system, because even small numbers (like 36) need such a lot of digits (100100=six digits) to represent them. What we really need is some system which maintains fairly close links to binary, the counting system of the computer, while at the same time being easier for us to use. One such numbering system is Hexadecimal, abbreviated to just Hex, which uses numbers to the base 16, i.e., there are sixteen different symbols in use (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F). You may be wondering how this can be like binary and yet be much more useful to use. The answer to this question is as follows. Each digit in Hex takes the place of four digits in binary as the following table shows:

Binary	*Hex*	*Decimal*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

This means that if we have some 8 bit binary number like 10100101 then it can be split up into two groups of 4 bits and each group can then be translated from the above table into Hex to give the Hex equivalent A5 (1010=A and 0101=5). In a Hex number each position to the left is 16 times the current position, starting as usual with units in the rightmost position. This means that the number A5H (the H after the number will be used throughout the rest of the book to show that a number is in Hex if this is not clear from the context) is actually AH lots of 16 plus 5 lots of one. As AH=10 in decimal, this becomes 10 lots of 16 plus 5 lots of one which equals 165 in decimal. The 8 bit number in binary that we started this example with, was 10100101, but we could have chosen any 8 bit number and you should see that we would always have ended with a two digit Hex value. You should now begin to see several advantages to using Hex numbers when dealing with computers. Some of these are: very easy conversion from binary to Hex and back; conversion from Hex to decimal is fairly straightforward; eight bit data values within micros can always be represented by two Hex digits; and although we have not shown it, 16 bit binary values such as addresses of memory locations can always be represented by 4 Hex digits.

As you can see, Hex numbers are far more convenient to use than decimal numbers when machine code is involved. But the only facilities available in BASIC for examining and modifying the contents of RAM memory are PEEK and POKE (from Chapter 3) and these two facilities both require their arguments to be in decimal. So if we are going to examine and enter the contents of RAM memory in Hex then some sort of translation is needed.

Below are listed two subroutines. One, starting at line 1000, converts a decimal integer in the range 0–65535 in variable D to its equivalent Hex value in the range 0–FFFF in variable H$. The second, starting at line 2000, converts a Hex number in H$ to its decimal equivalent in variable D.

```
1000 H$="" : FOR C=4 TO 1 STEP -1
1010 T=INT(D/16↑(C-1))-INT(D/16↑C)
     *16
1020 IF T>9 THEN 1040
1030 H$=H$+CHR$(T+48) : GOTO 1050
1040 H$=H$+CHR$(T+55)
1050 NEXT C : RETURN

2000 D=0 : FOR C=1 TO LEN(H$)
2010 D=D*16
2020 IF MID$(H$,C,1)>"9" THEN 2040
2030 D=D+ASC(MID$(H$,C,1))-48 :
     GOTO 2050
2040 D=D+ASC(MID$(H$,C,1))-55
2050 NEXT C : RETURN
```

If you study these two routines you should be able to discover how they operate. The only really tricky line is 1010:

T=INT(D/16↑(C−1))−INT(D/16↑C)*16

This line actually extracts the decimal value of each Hex digit in turn from the decimal number to be converted (in variable D), i.e. it extracts the Cth digit to the base 16 from the number contained in D. This line of code can in fact be generalised to the following:

T=INT(D/B↑(C−1))−INT(D/B↑C)*B

and it will now extract the Cth digit to any base B from the number contained in D.

A Machine Code Loader

Armed with these two subroutines, it is a fairly simple matter to go on and produce a program which will allow the examination

and modification of the contents of selected memory areas, which will give all of its outputs and responses in Hex. The program given below is such a loader and when you have entered it into your machine, come back and we'll discuss it more fully.

```
10 INPUT "ENTER START ADDRESS (HEX)"
   ;H$
20 IF H$="X" THEN END
30 GOSUB 2000 : K=D-1
40 K=K+1 : D=K : GOSUB 1000
50 PRINT H$;"  ";
60 D=PEEK(K) : GOSUB 1000
70 PRINT RIGHT$(H$,2);"  ";
80 H$="" : INPUT H$
90 IF H$="X" THEN 10
100 IF H$="" THEN 40
110 GOSUB 2000
120 POKE K,D
130 GOTO 40
1000 H$="" : FOR C=4 TO 1 STEP -1
1010 T=INT(D/16↑(C-1))-INT(D/16↑C)
     *16
1020 IF T>9 THEN 1040
1030 H$=H$+CHR$(T+48) : GOTO 1050
1040 H$=H$+CHR$(T+55)
1050 NEXT C : RETURN
2000 D=0 : FOR C=1 TO LEN(H$)
2010 D=D*16
2020 IF MID$(H$,C,1)>"9" THEN 2040
2030 D=D+ASC(MID$(H$,C,1))-48 :
     GOTO 2050
2040 D=D+ASC(MID$(H$,C,1))-55
2050 NEXT C : RETURN
```

If you enter and RUN this program then the first thing you will see is the message, ENTER START ADDRESS (HEX).

You should respond to this message by entering the address, in Hex, of the memory location whose contents you wish to examine. As an example let us enter the address 028A which is the address of a useful location in the system workspace. The program should cause the following output to appear on the VDU:

028A 00 ?

This shows the address that is being examined and the Hex value of its current contents, in this case zero.

The actual function of this memory location is to control the auto key repeat facility on the keyboard. You may have noticed that when you press and hold any of the cursor movement (arrow) keys on the keyboard that after a short delay the function of the key that you are holding is repeated at a fairly rapid rate. This feature is called auto key repeat. But from power on it only operates on certain keys. Changing the value of memory location 028A from 0 to 80 Hex causes auto key repeat to function on the whole keyboard so that the function of any key will be repeated after a short delay if you keep it pressed.

Going back to the display on the screen, i.e. 028A 00 ?

The question mark is the prompt printed by an INPUT statement (the one in line 80) and in fact the program is asking you whether or not you wish to change the value in the memory location displayed. If you wish to change the value then you should type in the new value in Hex and press return. If you do not wish to change the value, then just press return. Whichever of these two options you choose, pressing return will cause the next address to be displayed along with its contents and a question mark prompt to allow you to change these contents if you wish to. When you have finished examining or modifying one set of memory locations and the system is waiting for your response to the question mark prompt, you should type a capital X followed by pressing return. This will get you back to the ENTER START ADDRESS (HEX) message which you can answer

with a new address whose contents you wish to examine or you can answer this prompt with another X which will terminate this program altogether.

In the next chapter, we will present a complete Hex dump of a rather good extension to BASIC to give facilities for drawing lines and moving sprites and other High resolution graphics facilities, so if you have typed this program in, it might be a good idea to save it on tape, so that you have it if you want to enter and use the Extended BASIC facilities.

Chapter 10

THE MACHINE CODE ROUTINES

In this chapter we shall be presenting a Hex dump of a number of machine code graphics routines which we have written. The routines have been written so that they form an extension to the resident CBM BASIC and they are called by using their own keywords. Parameters are passed to these new keyword routines in normal BASIC integer variables. We think that this makes the routines much more friendly and easier to use.

The New Keywords

The Hex dump contains 9 new BASIC keywords, each of which is prefixed with the @ symbol. (The @ symbol forms part of the keyword and must not be left out.)

There now follows a description of each command and its function.

@OLD

@OLD restores a program which has been accidentally NEW'd. If @OLD is used when there is no program in memory (e.g. on switch on) then its action is indeterminate.

@MOVEBAS

@MOVEBAS moves the bottom of the BASIC program text area from address 2048 decimal up to address 16348 decimal. This protects the high resolution screen and frees space for 32 Sprites. Any program which is in memory will be lost and so @MOVEBAS prints a SURE? message before it moves the memory. A reply of Y will cause the bottom of BASIC to be moved. A reply of N (or anything except Y) will abort the command.

@HRG

@HRG locates the high resolution screen at address 8192 decimal and switches to high resolution mode.

@LRG

@LRG switches to low resolution (normal text) mode.

@GCLEAR

@GCLEAR clears the high resolution screen. (Provided it is located at address 8192 decimal.)

@GCOL

@GCOL sets up the foreground and background drawing colours on the high resolution screen. The foreground colour is read from the BASIC integer variable FC% and the background colour is read from BC%. If either FC% or BC% does not exist then normal text mode is entered and a SYNTAX ERROR message is printed. FC% and BC% are not checked for range, but only the low 4 bits of each variable are significant. (Numbers in the range 0 to 15.)

@STYLE

@STYLE sets up the drawing style which will be used for @PLOT and @LINE. The value of style is read from the BASIC integer variable S%. If S% is 0 then replace style is used. This means that a dot will be plotted or a line will be drawn regardless of the existing state of these dots on the screen. If S% is 1 then exclusive or style is used. This means that before a dot is plotted or a line is drawn, the existing state of the dots (1 to 0) is exclusive or'd with 1 to give the value which will be written.

For example, a dot which is plotted in exclusive or style will erase an existing dot at the same point or, if the dot was not already lit, it will light it.

If S% does not exist then normal mode is entered and a SYNTAX ERROR message is printed. If the value in S% is not 0 or 1 then normal mode is entered and an ILLEGAL QUANTITY ERROR message is printed.

@PLOT

@PLOT plots a point on the screen using the existing value of style. The X coordinate to be plotted is read from the BASIC integer variable X% and the Y coordinate is read from Y%. The system variables LASTX and LASTY are set to the values in X% and Y%. (See @LINE.) If either X% or Y% does not exist then normal text mode is entered and a SYNTAX ERROR message is printed. If either X% or Y% is off screen then the point will not be plotted.

@LINE

@LINE draws a line from the point in the system variables LASTX and LASTY (see @PLOT) to the point specified in the BASIC integer variables X% and Y%. The system variables LASTX and LASTY are then set to the values in X% and Y%. If either X% or Y% does not exist then normal text mode is entered and a SYNTAX ERROR message is printed. If a line goes off screen, then that part of the line which is on screen may or may not be drawn, depending on how far off screen it goes. Lines which are wholly on screen will always be drawn, so the rule is don't try to draw lines which go off screen!

All these commands can be used in either direct mode, or in program lines (either single or multi-statement lines). If they are used in direct mode however, the new keyword must be the first statement on the line or a SYNTAX ERROR will result.

About the Hex Dump

Once you have typed this lot in, you can save it by entering and RUNing the following program.

```
10 PRINT "{clr/home}"
20 FOR X=0 TO 1952 STEP 6 : PRINT
   X+100; "DATA ";
30 FOR Y=0 TO 5 : Z=PEEK(49152+X+Y)
40 PRINT MID$(STR$(Z),2); ", ";
50 NEXT Y : PRINT CHR$(20)
60 NEXT X : END
```

This will print lines of DATA statements on the screen. You can then simply place the cursor on each line in turn and enter it as though you had typed it yourself. When you have entered each DATA line, change the first 6 lines of the program to the following lines.

```
10 REM THIS LOADS THE EXTENDED BASIC
   KEYWORDS
20 FOR X=0 TO 1952
30 READ Z
40 POKE 49152+X, Z
50 NEXT X
60 END
```

You now have a BASIC program which, when RUN, will POKE the new keyword routines in for you. This program can be SAVEd to tape or disk under any name you like.

Before the new keywords can be used they must be linked in to the resident BASIC interpreter. This is done by typing SYS 49800 which links the new keywords in and print the following message on the screen.

```
**** CBM 64 EXTENDED BASIC ****
(C) AL CROSS 1983  *  64K RAM SYSTEM
READY.
```

The new keywords are now ready for use.

NOTE – Pressing RUN/STOP and RESTORE will disable the new keywords. They can be simply re-enabled by typing SYS 49800 again. (This will not damage any programs in memory.)

The Hex Dump

In order to save space we have listed the program with 8 bytes to a line. Every new line has the Hex address of the first byte printed alongside it so that you can keep track of where you have got to.

```
C000  4F 4C 44 00 4D 4F 56 45
C008  42 41 53 00 48 52 47 00
C010  4C 52 47 00 47 43 4C 45
C018  41 52 00 53 4D 4F 56 45
C020  00 53 54 59 4C 45 00 47
C028  43 4F 4C 00 50 4C 4F 54
C030  00 4C 49 4E 45 00 00 00
C038  00 00 00 00 00 00 00 00
C040  00 00 00 00 00 00 00 00
C048  00 00 00 00 00 00 00 00
C050  00 00 00 00 00 00 00 00
C058  00 00 00 00 00 00 00 00
C060  00 00 00 00 00 00 00 00
C068  00 00 00 00 00 00 00 00
C070  00 00 00 00 00 00 00 00
C078  00 00 00 00 00 00 00 00
C080  00 00 00 00 00 00 00 00
C088  00 00 00 00 00 00 00 00
C090  00 00 00 00 00 00 00 00
C098  00 00 00 00 00 00 00 00
C0A0  00 00 00 00 00 00 00 00
C0A8  00 00 00 00 00 00 00 00
C0B0  00 00 00 00 00 00 00 00
C0B8  00 00 00 00 00 00 00 00
C0C0  00 00 00 00 00 00 00 00
C0C8  00 00 00 00 00 00 00 00
C0D0  00 00 00 00 00 00 00 00
C0D8  00 00 00 00 00 00 00 00
C0E0  00 00 00 00 00 00 00 00
```

```
C0E8  00 00 00 00 00 00 00 00
C0F0  00 00 00 00 00 00 00 00
C0F8  00 00 00 00 00 00 00 00
C100  45 C3 99 C3 D6 C3 E7 C3
C108  F8 C3 69 C4 24 C5 45 C5
C110  5B C5 6B C6 40 C3 40 C3
C118  40 C3 40 C3 40 C3 40 C3
C120  40 C3 40 C3 40 C3 40 C3
C128  40 C3 40 C3 40 C3 40 C3
C130  40 C3 40 C3 40 C3 40 C3
C138  40 C3 40 C3 40 C3 40 C3
C140  40 C3 40 C3 40 C3 40 C3
C148  40 C3 40 C3 40 C3 40 C3
C150  40 C3 40 C3 40 C3 40 C3
C158  40 C3 40 C3 40 C3 40 C3
C160  40 C3 40 C3 40 C3 40 C3
C168  40 C3 40 C3 40 C3 40 C3
C170  40 C3 40 C3 40 C3 40 C3
C178  40 C3 40 C3 40 C3 40 C3
C180  40 C3 40 C3 40 C3 40 C3
C188  40 C3 40 C3 40 C3 40 C3
C190  40 C3 40 C3 40 C3 40 C3
C198  40 C3 40 C3 40 C3 40 C3
C1A0  40 C3 40 C3 40 C3 40 C3
C1A8  40 C3 40 C3 40 C3 40 C3
C1B0  40 C3 40 C3 40 C3 40 C3
C1B8  40 C3 40 C3 40 C3 40 C3
C1C0  40 C3 40 C3 40 C3 40 C3
C1C8  40 C3 40 C3 40 C3 40 C3
C1D0  40 C3 40 C3 40 C3 40 C3
C1D8  40 C3 40 C3 40 C3 40 C3
C1E0  40 C3 40 C3 40 C3 40 C3
C1E8  40 C3 40 C3 40 C3 40 C3
```

continued

```
C1F0  40 C3 40 C3 40 C3 40 C3
C1F8  40 C3 40 C3 40 C3 40 C3
C200  93 0D 20 20 20 20 20 2A
C208  2A 2A 20 43 42 4D 36 34
C210  20 2D 20 45 58 54 45 4E
C218  44 45 44 20 42 41 53 49
C220  43 20 2A 2A 2A 0D 0D 0A
C228  20 20 28 43 29 20 41 4C
C230  20 43 52 4F 53 53 20 31
C238  39 38 33 20 20 2A 20 20
C240  36 34 4B 20 52 41 4D 20
C248  53 59 53 54 45 4D 0D 0D
C250  0A 52 45 41 44 59 2E 0D
C258  0A 00 00 FF FF FF 9D DF
C260  00 62 00 20 00 00 00 00
C268  00 40 00 42 00 42 62 02
C270  00 80 00 02 00 00 42 00
C278  60 00 00 00 66 00 FF 00
C280  FD 7D FF F7 FF FF FF FF
C288  A9 00 A0 C2 20 1E AB A9
C290  4C 85 7C A9 9E 85 7D A9
C298  C2 85 7E 6C 02 03 C9 40
C2A0  D0 44 A5 9D F0 28 AD 00
C2A8  02 C9 40 D0 1C 20 F8 C2
C2B0  A0 00 B1 7A C9 20 F0 09
C2B8  E6 7A D0 F6 E6 7B 38 B0
C2C0  F1 20 74 A4 A9 00 38 B0
C2C8  1D A9 40 38 B0 18 20 F8
C2D0  C2 A0 00 B1 7A C9 00 F0
C2D8  0D C9 3A F0 09 E6 7A D0
C2E0  F2 E6 7B 38 B0 ED C9 3A
C2E8  B0 0A C9 20 F0 07 38 E9
C2F0  30 38 E9 D0 60 4C 73 00
C2F8  A9 00 85 7F A9 C0 85 80
C300  E6 7A D0 02 E6 7B A0 00
```

```
C308  A2 00 B1 7F F0 24 D1 7A
C310  D0 04 C8 38 B0 F4 B1 7F
C318  F0 04 C8 38 B0 F8 C8 98
C320  18 65 7F 85 7F A9 00 65
C328  80 85 80 A0 00 E8 E8 38
C330  B0 D8 BD 00 C1 85 80 E8
C338  BD 00 C1 85 81 6C 80 00
C340  A2 0B 6C 00 03 A5 2B 85
C348  86 A5 2C 85 87 A0 04 A9
C350  00 D1 86 F0 04 C8 38 B0
C358  F8 C8 98 A0 00 18 65 86
C360  91 86 C8 A9 00 65 87 91
C368  86 A0 00 B1 86 D0 1B C8
C370  B1 86 D0 16 A9 02 18 65
C378  86 85 2D 85 2F 85 31 A9
C380  00 65 87 85 2E 85 30 85
C388  32 60 A0 00 B1 86 AA C8
C390  B1 86 85 87 86 86 38 B0
C398  D0 A9 CF A0 C3 20 1E AB
C3A0  20 CF FF C9 59 D0 27 A9
C3A8  00 8D 00 40 8D 01 40 8D
C3B0  02 40 8D 81 02 A9 40 85
C3B8  2C 85 2E 85 30 85 32 8D
C3C0  82 02 A9 03 85 2D 85 2F
C3C8  85 31 A9 01 85 2B 60 0D
C3D0  53 55 52 45 3F 00 A9 08
C3D8  0D 18 D0 8D 18 D0 A9 20
C3E0  0D 11 D0 8D 11 D0 60 A9
C3E8  F7 2D 18 D0 8D 18 D0 A9
C3F0  DF 2D 11 D0 8D 11 D0 60
C3F8  A9 00 85 86 A9 20 85 87
C400  A9 40 85 88 A9 3F 85 89
C408  A9 00 85 8A A5 87 C5 89
```

continued

```
C410  D0 07 A5 86 C5 88 D0 01
C418  60 A0 00 A5 8A 91 86 E6
C420  86 D0 E9 E6 87 38 B0 E4
C428  A5 2D 85 86 A5 2E 85 87
C430  A0 00 A5 87 C5 30 D0 08
C438  A5 86 C5 2F D0 02 38 60
C440  B1 86 C5 88 D0 11 C8 B1
C448  86 C5 89 D0 0B C8 B1 86
C450  AA C8 B1 86 A8 18 60 C8
C458  98 18 69 06 65 86 85 86
C460  A5 87 69 00 85 87 38 B0
C468  C7 A9 D3 85 88 A9 CE 85
C470  89 20 28 C4 B0 57 98 30
C478  57 C9 08 B0 53 0A 85 FB
C480  A9 01 C8 88 F0 04 0A 38
C488  B0 F9 85 8A A9 D9 85 88
C490  A9 80 85 89 20 28 C4 B0
C498  34 A6 FB 98 9D 01 D0 A9
C4A0  D8 85 88 A9 80 85 89 20
C4A8  28 C4 B0 21 8A 48 A6 FB
C4B0  98 9D 00 D0 68 C9 00 D0
C4B8  0B A5 8A 49 FF 2D 10 D0
C4C0  8D 10 D0 60 A5 8A 0D 10
C4C8  D0 8D 10 D0 60 4C D3 C4
C4D0  4C DB C4 20 E3 C4 A2 0B
C4D8  6C 00 03 20 E3 C4 A2 0E
C4E0  6C 00 03 A9 1B 8D 11 D0
C4E8  A9 15 8D 18 D0 60 A9 C2
C4F0  85 88 A9 C3 85 89 20 28
C4F8  C4 B0 22 8A D0 22 98 29
C500  0F 85 8A A9 C6 85 88 A9
C508  C3 85 89 20 28 C4 B0 0D
C510  8A D0 0D 98 0A 0A 0A 0A
C518  05 8A 85 8A 60 4C D3 C4
C520  4C DB C4 00 A9 D3 85 88
```

```
C528  A9 80 85 89 20 28 C4 B0
C530  0E 8A D0 0E 98 F0 04 C9
C538  01 D0 07 8C 23 C5 60 4C
C540  D3 C4 4C DB C4 20 EE C4
C548  A9 00 85 86 A9 04 85 87
C550  A9 E8 85 88 A9 07 85 89
C558  4C 0C C4 20 AA C5 20 8C
C560  C5 A5 FB 8D 87 C7 A5 FC
C568  8D 88 C7 A5 FD 8D 89 C7
C570  20 8C C5 20 D0 C5 A0 00
C578  48 AD 23 C5 C9 01 F0 06
C580  68 11 FB 38 B0 03 68 51
C588  FB 91 FB 60 A5 FC C9 01
C590  F0 04 B0 13 90 06 A5 FB
C598  C9 40 B0 0B A5 FE D0 07
C5A0  A5 FD C9 C8 B0 01 60 68
C5A8  68 60 A9 D8 85 88 A9 80
C5B0  85 89 20 28 C4 B0 16 86
C5B8  FC 84 FB A9 D9 85 88 A9
C5C0  80 85 89 20 28 C4 B0 05
C5C8  84 FD 86 FE 60 4C D3 C4
C5D0  A5 FB 48 29 07 8D 18 C6
C5D8  68 29 F8 85 FB A5 FD 48
C5E0  29 07 8D 19 C6 68 29 F8
C5E8  48 4A 4A 4A 85 FD 4A 4A
C5F0  18 65 FD 85 FD 68 0A 0A
C5F8  0A 0D 19 C6 18 65 FB 85
C600  FB A5 FD 65 FC 18 69 20
C608  85 FC A9 80 AE 18 C6 E8
C610  CA F0 04 4A 38 B0 F9 60
C618  00 00 AD 91 C7 38 ED 93
C620  C7 8D 91 C7 AD 92 C7 ED
C628  94 C7 8D 92 C7 08 F0 0C
```

continued

```
C630  10 05 A9 FF 38 B0 0E A9
C638  01 38 B0 09 AD 91 C7 C9
C640  00 D0 F4 A9 00 AA 28 10
C648  1B AD 92 C7 49 FF 8D 92
C650  C7 AD 91 C7 49 FF 18 69
C658  01 8D 91 C7 A9 00 6D 92
C660  C7 8D 92 C7 AC 92 C7 AD
C668  91 C7 60 20 AA C5 A5 FB
C670  8D 84 C7 8D 91 C7 A5 FC
C678  8D 85 C7 8D 92 C7 A5 FD
C680  8D 86 C7 AD 87 C7 8D 93
C688  C7 AD 88 C7 8D 94 C7 20
C690  1A C6 8E 8F C7 8C 8C C7
C698  8D 8B C7 A5 FD 8D 91 C7
C6A0  A9 00 8D 92 C7 AD 89 C7
C6A8  8D 93 C7 A9 00 8D 94 C7
C6B0  20 1A C6 8E 90 C7 8C 8E
C6B8  C7 8D 8D C7 AD 87 C7 85
C6C0  FB AD 88 C7 85 FC AD 89
C6C8  C7 85 FD 20 70 C5 AD 8D
C6D0  C7 8D 95 C7 AD 8E C7 8D
C6D8  96 C7 AD 96 C7 30 45 D0
C6E0  05 AD 95 C7 F0 3E 38 AD
C6E8  95 C7 ED 8B C7 8D 95 C7
C6F0  AD 96 C7 ED 8C C7 8D 96
C6F8  C7 AD 90 C7 F0 1A 30 0B
C700  EE 89 C7 D0 13 EE 8A C7
C708  38 B0 0D CE 89 C7 AD 89
C710  C7 C9 FF D0 03 CE 8A C7
C718  AD 96 C7 30 07 D0 37 AD
C720  95 C7 D0 32 18 AD 95 C7
C728  6D 8D C7 8D 95 C7 AD 96
C730  C7 6D 8E C7 8D 96 C7 AD
C738  8F C7 F0 1A 30 0B EE 87
C740  C7 D0 13 EE 88 C7 38 B0
```

```
C748  0D CE 87 C7 AD 87 C7 C9
C750  FF D0 03 CE 88 C7 AD 87
C758  C7 85 FB AD 88 C7 85 FC
C760  AD 89 C7 85 FD 20 70 C5
C768  AD 87 C7 CD 84 C7 D0 11
C770  AD 88 C7 CD 85 C7 D0 09
C778  AD 89 C7 CD 86 C7 D0 01
C780  60 4C DA C6 00 00 00 00
```

OTHER BOOKS OF INTERST

BP86: AN INTRODUCTION TO BASIC PROGRAMMING TECHNIQUES
S. Daly, M.B.C.S.

This book is based on the author's own experience in learning BASIC and in helping others, mostly beginners, to program and understand the language.

Also included is a program library containing various programs that the author has actually written and run – these are for biorhythms, plotting a graph of y and x , standard deviation, regression, generating a musical note sequence and a card game.

The book is completed by a number of appendices which include test questions and answers on each chapter and a glossary.

96 pages *1981*
0 85934 061 9 £1.95

BP115: THE PRE-COMPUTER BOOK
F.A. Wilson, C.G.I.A., C.Eng., F.I.E.E., F.I.E.R.E., F.B.I.M.

Aimed at the absolute beginner with no knowledge of computing, this entirely non-technical discussion of computer bits and pieces and programming is written mainly for those who do not possess a microcomputer but either intend to one day own one or simply wish to know something about them.

Also highly recommended for the new computer owner who may be beset with uncertainties and, also, the person who cannot understand the jargon and technical terms used by most manufacturers in their sales leaflets.

96 pages *1983*
0 85934 090 2 £1.95

BP126: BASIC & PASCAL IN PARALLEL
S.J. Wainwright, B.Sc., Ph.D., M.I.Biol.

This book takes the two languages BASIC and Pascal, and develops programs in both languages simultaneously. Emphasis is placed on structured programming by the systematic use of control structures; and modular program design is used throughout. Example programs are

used to illustrate the program structures as they are introduced, and the reader can learn by example.

If the book is used as an introduction to BASIC programming, the structured approach will encourage good programming techniques which will be compatible with Pascal programming at a later date, and will eliminate many of the difficulties met by BASIC programmers when they move over to programming in Pascal, and have to rethink their approach to program design. BASIC is used to emulate Pascal throughout the text, so the transition from BASIC to Pascal should present few problems to a person who can already program in BASIC.

If the book is used as an introduction to Pascal programming, the presence of equivalent programs in BASIC but having Pascal-like structure, will provide a familiar BASIC "handle" with which to grasp the Pascal programming techniques. If the reader does not yet possess a Pascal interpreter or compiler, he/she can learn many of the features of Pascal by using the BASIC programs on his/her own computer, and comparing them with the Pascal listings.

The common ground between the BASIC and Pascal programming languages is covered, and emphasis is placed on the similarities rather than the differences between them. As the title suggests, the book is intended as a bilingual introduction to programming which can be used to learn programming in both languages simultaneously, and to learn programming techniques which are compatible with both languages.

64 pages *1983*
0 85934 101 1 **£1.50**

BP137: BASIC & FORTRAN IN PARALLEL
S.J. Wainright, B.Sc., Ph.D., M.I.Biol. & A. Grant, B.Sc.

BASIC (Beginners Allpurpose Symbolic Instruction Code) was developed in the mid 1960s as a language which would be easy to learn and use. It is available on almost all microcomputers, often residing in a read only memory, so that it is available as soon as the machine is switched on.

FORTRAN is an acronym for FORmula TRANslation. It was developed in the mid 1950's and was one of the first high level languages to be developed, and the first to be used extensively. This historical accident has meant that FORTRAN occupies a position of importance; with much programming, particularly for scientific

purposes, being done in FORTRAN. It is unlikely to be dislodged from this position in the near future, if ever, as so much time and money has been invested in writing FORTRAN programs. At the time of writing, FORTRAN is available on at least 70 microcomputers, and this number is likely to increase. As an appendix to this book, we have included a FORTRAN interpreter written in Sinclair Spectrum BASIC, which should also run with little modification on the Sinclair ZX-81. This supports most of the commonly used features of FORTRAN, and makes it possible to "get a feel" of what writing programs in FORTRAN is like. We strongly recommend that if the reader has a Sinclair Spectrum computer available, he/she types in and saves the FORTRAN interpreter before reading any further in this book. If this is done, many of the features of FORTRAN discussed in the book can be practised as they are encountered.

The FORTRAN programs presented in this book were developed using FORTRAN 4 on a PRIME 750 computer, and the BASIC programs were developed using a 16K Sinclair Spectrum and a Sharp MZ80 microcomputer.

This book could be used to learn FORTRAN, it could be used to learn BASIC. It could also be used to learn both at the same time. Using the FORTRAN interpreter presented in the appendix, it will be possible for the reader to learn at least a subset of FORTRAN, even if he/she does not possess a full FORTRAN compiler.

96 pages *1984*
0 85934 112 7 £1.95

BP138: BASIC & FORTH IN PARALLEL
S.J. Wainwright, B.Sc., Ph.D., M.I.Biol

As in books BP126 and BP137 the computer languages BASIC and Forth are this time considered simultaneously, again the emphasis being placed on the common ground between them rather than their differences.

An ideal book for those who are familiar with BASIC and want to learn Forth or for those who just want to start getting into Forth.

Also see books BP126 and BP137.

96 pages *1984*
0 85934 113 5 £1.95

Please note following is a list of other titles that are available in our range of Radio, Electronics and Computer Books.

These should be available from all good Booksellers, Radio Component Dealers and Mail Order Companies.

However, should you experience difficulty in obtaining any title in your area, then please write directly to the publisher enclosing payment to cover the cost of the book plus adequate postage.

If you would like a complete catalogue of our entire range of Radio, Electronics and Computer Books then please send a Stamped Addressed Envelope to:

BERNARD BABANI (publishing) LTD
THE GRAMPIANS
SHEPHERDS BUSH ROAD
LONDON W6 7NF
ENGLAND

BP78	Practical Computer Experiments	£1.75
BP79	Radio Control for Beginners	£1.75
BP80	Popular Electronic Circuits – Book 1	£1.95
BP81	Electronic Synthesiser Projects	£1.75
BP82	Electronic Projects Using Solar Cells	£1.95
BP83	VMOS Projects	£1.95
BP84	Digital IC Projects	£1.95
BP85	International Transistor Equivalents Guide	£2.95
BP86	An Introduction to BASIC Programming Techniques	£1.95
BP87	Simple L.E.D. Circuits – Book 2	£1.35
BP88	How to Use Op-Amps	£2.25
BP89	Elements of Electronics – Book 5	£2.95
BP90	Audio Projects	£1.95
BP91	An Introduction to Radio DXing	£1.95
BP92	Easy Electronics – Crystal Set Construction	£1.75
BP93	Electronic Timer Projects	£1.95
BP94	Electronic Projects for Cars and Boats	£1.95
BP95	Model Railway Projects	£1.95
BP96	C B Projects	£1.95
BP97	IC Projects for Beginners	£1.95
BP98	Popular Electronic Circuits – Book 2	£2.25
BP99	Mini-Matrix Board Projects	£1.95
BP100	An Introduction to Video	£1.95
BP101	How to Identify Unmarked IC's	£0.65
BP102	The 6809 Companion	£1.95
BP103	Multi-Circuit Board Projects	£1.95
BP104	Electronic Science Projects	£2.25
BP105	Aerial Projects	£1.95
BP106	Modern Op-Amp Projects	£1.95
BP107	30 Solderless Breadboard Projects – Book 1	£2.25
BP108	International Diode Equivalents Guide	£2.25
BP109	The Art of Programming the 1K ZX81	£1.95
BP110	How to Get Your Electronic Projects Working	£1.95
BP111	Elements of Electronics – Book 6	£3.50
BP112	A Z-80 Workshop Manual	£2.75
BP113	30 Solderless Breadboard Projects – Book 2	£2.25
BP114	The Art of Programming the 16K ZX81	£2.50
BP115	The Pre-Computer Book	£1.95
BP116	Electronic Toys Games & Puzzles	£2.25
BP117	Practical Electronic Building Blocks – Book 1	£1.95
BP118	Practical Electronic Building Blocks – Book 2	£1.95
BP119	The Art of Programming the ZX Spectrum	£2.50
BP120	Audio Amplifier Fault-Finding Chart	£0.65
BP121	How to Design and Make your Own P.C.B.s	£1.95
BP122	Audio Amplifier Construction	£2.25
BP123	A Practical Introduction to Microprocessors	£2.25
BP124	Easy Add-on Projects for Spectrum ZX81 & Ace	£2.75
BP125	25 Simple Amateur Band Aerials	£1.95
BP126	BASIC & PASCAL in Parallel	£1.50
BP127	How to Design Electronic Projects	£2.25
BP128	20 Programs for the ZX Spectrum & 16K ZX81	£1.95
BP129	An Introduction to Programming the ORIC-1	£1.95
BP130	Micro Interfacing Circuits – Book 1	£2.25
BP131	Micro Interfacing Circuits – Book 2	£2.25
BP132	25 Simple Shortwave Broadcast Band Aerials	£1.95
BP133	An Introduction to Programming the Dragon 32	£1.95
BP134	Easy Add-on Projects for Commodore 64 & Vic-20	£2.50
BP135	The Secrets of the Commodore 64	£2.50
BP136	25 Simple Indoor and Window Aerials	£1.95
BP137	BASIC & FORTRAN in Parallel	£1.95
BP138	BASIC & FORTH in Parallel	£1.95
BP139	An Introduction to Programming the BBC Model B Micro	£2.50
BP140	Digital IC Equivalents and Pin Connections	£3.95
BP141	Linear IC Equivalents and Pin Connections	£3.95
BP142	An Introduction to Programming the Acorn Electron	£1.95
BP143	An Introduction to Programming the Atari 600XL and 800XL	£2.50
BP144	Further Practical Electronics Calculations and Formulae	£3.75
BP145	25 Simple Tropical and M.W. Band Aerials	£1.95

160	Coil Design and Construction Manual	£1.95
202	Handbook of Integrated Circuits (IC's) Equivalents & Substitutes	£1.95
205	First Book of Hi-Fi Loudspeaker Enclosures	£0.95
208	Practical Stereo and Quadrophony Handbook	£0.75
214	Audio Enthusiasts Handbook	£0.85
219	Solid State Novelty Projects	£0.85
220	Build Your Own Solid State Hi-Fi and Audio Accessories	£0.85
221	28 Tested Transistor Projects	£1.25
222	Solid State Short Wave Receivers for Beginners	£1.95
223	50 Projects Using IC CA3130	£1.25
224	50 CMOS IC Projects	£1.35
225	A Practical Introduction to Digital IC's	£1.75
226	How to Build Advanced Short Wave Receivers	£1.95
227	Beginners Guide to Building Electronic Projects	£1.95
228	Essential Theory for the Electronics Hobbyist	£1.95
RCC	Resistor Colour Code Disc	£0.20
BP1	First Book of Transistor Equivalents and Substitutes	£1.50
BP2	Handbook of Radio, TV, Ind & Transmitting Tube & Valve Equivalents	£0.60
BP6	Engineers and Machinists Reference Tables	£0.75
BP7	Radio and Electronic Colour Codes and Data Chart	£0.40
BP14	Second Book of Transistor Equivalents & Substitutes	£1.75
BP24	52 Projects Using IC741	£1.75
BP27	Chart of Radio, Electronic, Semi-conductor and Logic Symbols	£0.50
BP28	Resistor Selection Handbook	£0.60
BP29	Major Solid State Audio Hi-Fi Construction Projects	£0.85
BP32	How to Build Your Own Metal and Treasure Locators	£1.95
BP33	Electronic Calculator Users Handbook	£1.50
BP34	Practical Repair and Renovation of Colour TVs	£1.95
BP36	50 Circuits Using Germanium, Silicon and Zener Diodes	£1.50
BP37	50 Projects Using Relays, SCR's and TRIACS	£1.95
BP39	50 (FET) Field Effect Transistor Projects	£1.75
BP42	50 Simple L.E.D. Circuits	£1.50
BP43	How to Make Walkie-Talkies	£1.95
BP44	IC 555 Projects	£1.95
BP45	Projects in Opto-Electronics	£1.95
BP47	Mobile Discotheque Handbook	£1.35
BP48	Electronic Projects for Beginners	£1.95
BP49	Popular Electronic Projects	£1.95
BP50	IC LM3900 Projects	£1.35
BP51	Electronic Music and Creative Tape Recording	£1.95
BP52	Long Distance Television Reception (TV-DX) for the Enthusiast	£1.95
BP53	Practical Electronics Calculations and Formulae	£2.95
BP54	Your Electronic Calculator and Your Money	£1.35
BP55	Radio Stations Guide	£1.75
BP56	Electronic Security Devices	£1.95
BP57	How to Build Your Own Solid State Oscilloscope	£1.95
BP58	50 Circuits Using 7400 Series IC's	£1.75
BP59	Second Book of CMOS IC Projects	£1.95
BP60	Practical Construction of Pre-amps, Tone Controls, Filters & Attenuators	£1.95
BP61	Beginners Guide To Digital Techniques	£0.95
BP62	Elements of Electronics – Book 1	£2.95
BP63	Elements of Electronics – Book 2	£2.25
BP64	Elements of Electronics – Book 3	£2.25
BP65	Single IC Projects	£1.50
BP66	Beginners Guide to Microprocessors and Computing	£1.95
BP67	Counter, Driver and Numeral Display Projects	£1.75
BP68	Choosing and Using Your Hi-Fi	£1.65
BP69	Electronic Games	£1.75
BP70	Transistor Radio Fault-Finding Chart	£0.50
BP71	Electronic Household Projects	£1.75
BP72	A Microprocessor Primer	£1.75
BP73	Remote Control Projects	£1.95
BP74	Electronic Music Projects	£1.75
BP75	Electronic Test Equipment Construction	£1.75
BP76	Power Supply Projects	£1.95
BP77	Elements of Electronics – Book 4	£2.95

BERNARD BABANI BP135

Secrets of the COMMODORE 64

■ This book is intended as a beginner's guide to the Commodore 64. In it, we will be giving you masses of useful information and programming tips as well as describing how to get the best from the powerful sound and graphics facilities. The only assumption we have made is that you already have a copy of the Commodore 64 User's Manual (which comes free with the computer anyway!).

■ The book gets under way in Chapter 1 with a brief look at the way the memory is organised and how much of it is available to you from BASIC. Chapter 2 is all about random numbers and ways of generating them. This is important because random numbers have endless roles to play in both entertainment and serious programming applications. Chapter 3 is another general chapter, this time on two very useful and often misunderstood BASIC facilities – PEEK and POKE. Our look at the Commodore 64's special facilities begins in Chapters 4 and 5 with a description of low resolution graphics, colour and simple animation. Chapter 6 takes us into the more advanced world of animation and looks at Sprites. Chapter 7 explains the mysteries of high resolution graphics, including plotting points and drawing lines and circles. Chapter 8 describes the use of the fabulous sound facilities, including using them for both music and sound effects. And to finish we have two very useful chapters dealing with machine code. Chapter 9 is concerned with machine code programs in general on the Commodore 64, and then Chapter 10 presents a machine code program which extends the BASIC on the machine by adding nine additional statement keywords which can be used in any BASIC program.

GB £ NET +001.95

ISBN 0-85934-110-0

£1.95

00195

9 780859 341103