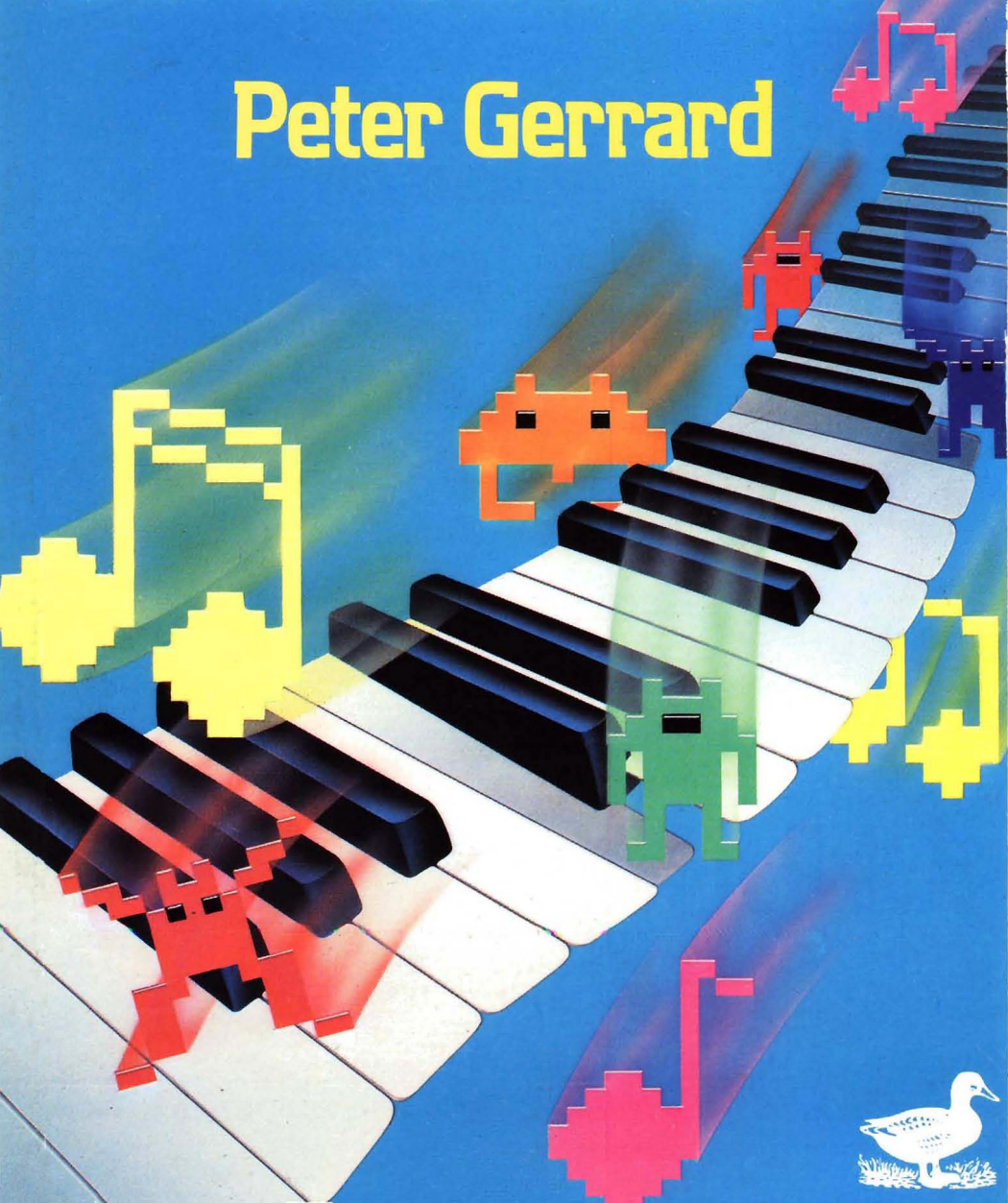


# SPRITES & SOUND ON THE COMMODORE 64

Peter Gerrard





# **SPRITES & SOUND ON THE 64**

**Peter Gerrard**



**Duckworth**

First published in 1984 by  
Gerald Duckworth & Co. Ltd.  
The Old Piano Factory  
43 Gloucester Crescent, London NW1

©1984 by Peter Gerrard

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

ISBN 0 7156 1781 8

British Library Cataloguing in Publication Data

Gerrard, Peter

Sprites & Sound on the 64.

(Duckworth home computing)

1. Computer graphics 2. Commodore 64  
(Computer)

I. Title

001.64'43 T385

ISBN 0-7156-1781-8

Typeset by The Electronic Village, Richmond  
from text stored on a Commodore 64  
Printed in Great Britain by  
Redwood Burn Ltd., Trowbridge  
and bound by Pegasus Bookbinding, Melksham

# Contents

Preface	7
1. 64 Graphics and Sound	9
2. General Introduction to Graphics	15
3. Using the Existing Graphics Set	25
4. Sprites	43
5. User-Defined Graphics	95
6. High Resolution Graphics	121
7. General Introduction to Sound	139
8. Starting to Play with Sound	151
9. Further Sound Techniques	179
10. Adding Commands to Basic	203
Appendix: An assembler/disassembler	223
Index	254

## **SPRITES & SOUND ON THE 64**

The 13 longest programs in  
this book are available on one  
cassette at £7.95.

They are available only  
from the publisher.

Send your cheque/  
postal order to:

**DUCKWORTH**  
The Old Piano Factory  
43 Gloucester Crescent  
London NW1

and they will be sent to you  
post-free.

# Preface

Unlike many books on the Commodore 64, this one is concerned purely with the use of graphics and sound. Only two diversions occur, and these are a brief look at adding commands to the existing Commodore Basic command set, in order to improve upon that Basic, and a disassembly for a machine code assembler/disassembler which you can use to speed up machine code program development time.

Although the longest chapter in this book is indeed concerned with sprites, as the title would suggest, all aspects of using graphics on the Commodore 64 are looked at in some detail, including producing user defined characters and using and manipulating high resolution graphics.

The sections on sound cover such advanced topics as ring modulation and filtering, as well as taking more than a passing look at all the other features that make the Commodore 64 such an excellent computer to 'have a play around with'.

By the end of the book, readers should be much more familiar, and at home with, the use of both graphics and sound, and be much better equipped to incorporate both of these outstanding features into their own programs.

Throughout the book there are numerous example listings, illustrating the topics under discussion in each section, and some of the programs included cover such items as designing and creating data statements for multi-coloured sprites, turning the keyboard into a synthesiser (and introducing the reader to musical effects such as glissando and synchronising notes), a delightful piece of animation using the existing character set, and an extremely useful assembler/disassembler for the Commodore 64.

All the lengthier programs have been gathered together on one cassette for those who have neither the time nor the patience to type them all in from the listings given. This cassette is available direct from the publishers.

I'd like to thank everyone who has directly or indirectly contributed

anything to this book, but if any mistakes remain they are purely my own: blame me!

Finally, thanks to mum and dad. I know you probably wouldn't know what a computer was if it fell on your head, but who cares ? Keep up the editing, and keep up the gardening, and say hello to everyone 'up north'. A'reet ?

P.G.



# 64 Graphics and Sound

## **The strengths and weaknesses of the 64**

As a home computer, the Commodore 64 stands out from most of the others currently available because of the quality of both its graphics and its sound.

On the graphics side, the standard Commodore 64 can handle sprites (both single colour and multi-colour), user defined characters, single pixel addressing of the screen, and has built into it a total of 16 different colours which can be displayed at any one time.

By the way, some of these terms may not mean too much to you at present, but we'll come to them all in time!

With sound, the 64 has the ability to emulate many custom built musical synthesisers, with three independent voices covering an eight octave range, each voice capable of being played in any one of four different wave forms: pulse, sawtooth, triangle, and noise.

With features such as these, you'd be readily forgiven for thinking that the Commodore 64 has also got built into it a superb version of the programming language Basic, in order to handle all these outstanding capabilities.

Well, there the Commodore 64 and the rest of the world part company, because the version of Basic installed in it is primitive to say the least. There are no commands devoted to graphics, and there are no commands devoted to sound, unlike most other computers that attempt to compete, which come equipped with a wealth of commands like FILL, PLOT, ENVELOPE, and so on.

Apart from these obvious criticisms of the version of Basic, there are many others that could be levelled at it. Structuring is virtually impossible, there is no PRINT USING, procedures cannot be defined, and there are plenty more deficiencies where those come from.

Presumably the reasons behind the resignation in January 1984 of Jack Tramiel, the company supremo who started the whole thing off in the first place, were to do with things other than the level of Basic used in his computers. However, if he'd been a programmer that would have figured very highly indeed!

## **Back to basics**

Despite this, given what we have got in the Commodore 64 there is still a lot that we can do simply from Basic. It's just that it isn't exactly easy, since there are only two commands that we can use to affect any of the registers in the Commodore 64, namely PEEK and POKE.

As you probably know, POKE is used to affect the content of a memory location, be that location on the screen, in the Commodore 64's video chip, or wherever. PEEK simply tells us what is stored at any location at any one time, although please note that some locations are what is termed 'Read Only', and will return misleading values if PEEKed. Like all companies, Commodore can be a bit secretive about how their machines do what they do.

Later on in this book we'll be going into more detail about all the technical terms encountered, explaining both what they mean, and how they are used.

Since we can only use these two commands some of the operations we'd like to perform - for example wiping out an area of memory in order to use it to display a high resolution screen - can take quite some time. However, this is only true if we remain with Basic, and in one section of this book we'll be giving you a set of machine code routines to perform some simple tasks with high resolution plotting and other graphical functions.

As mentioned earlier, one of the reasons why using both graphics and sound on the 64 is so difficult, is its version of the programming language Basic that we have to come to grips with. There are a number of packages already on the market that attempt to overcome these difficulties by the straightforward process of simply adding more commands to the repertoire that you already have.

But these packages cost money, in at least one case a great deal of money, in order to correct something that should never have been wrong in the first place.

How much better it would be if you, the person actually using the machine, could correct these faults yourself. Commodore's own manuals are not at their best in this area, and so the last part of this book is devoted to explaining just one way in which you could easily add commands to the existing command set. There are a couple of commands there to get you started, and after reading through the book you should be in a position to add many more of your own.

The commands could be added as words (e.g. PLOT), symbols (e.g. '\*'), but as we're generously given a set of function keys at the right-hand side of the keyboard, why not use them? No one else seems to, so one of the programs in the final chapter will redefine all the function keys to be something useful, and in a couple of instances something unusual as well.

### **A word of explanation**

If all this seems to be taking us away from the aims expressed in the title of the book, don't panic (as they say). Sprites and sound on the Commodore 64 are indeed well covered here, but we couldn't write a book about graphics and music without documenting all the other wonderful features of the machine, such as high resolution plotting and redefining the existing character set. Publishers just like alliterative titles, that's all!

Similarly, as it is so difficult to do everything in Basic using only what we're given with the 64, it makes sense to dive into machine code to perform some of the tasks that would take an eternity in Basic. If you're unfamiliar with machine code, well, there's no great need to worry. Everything is covered at a relatively gentle pace, and just to help you out the final chapter of the book is a disassembly of an assembler/monitor for the 64, which should make machine code life a lot easier.

If the thought of typing it all in is a daunting one, that and all the other major programs in this book are available on cassette from the publishers at nominal cost. If you do type it in and can't get it working, and then insist that there's something wrong with the listing, bear in mind a simple fact: how do you think we're able to sell a working copy of it on tape if it's incorrect? The program was used to list itself!

## Some definitions

Before we enter into the nitty-gritty of it all and find ourselves merrily altering characters and moving multi-coloured sprites about the screen without understanding a word of what's going on, let's get a few definitions out of the way first.

These are all terms that you'll encounter throughout the rest of the book, so if we can assume a working knowledge of Commodore's version of Basic (although we don't expect you to be the world's greatest programmer!), let's start with decimal and hexadecimal.

We're all used to the numerical system of counting as used in the majority of the western world. This relies on the numeric symbols 0 to 9, and since this gives us ten symbols in all to play with, we describe this system as having a numerical base of ten. Using those ten symbols we can then group them together to express larger numbers: 567, 1938, and so on.

What does the number 1938 really represent? Remembering that we're using a base of ten, the number 1938 can be thought of as being 8 times 10 to the power zero (which is mathematically defined to be equal to 1), plus 3 times 10 to the power 1, plus 9 times 10 to the power 2, plus 1 times 10 to the power 3. This is equal to 8, plus 30, plus 900, plus 1000, or in other words 1938.

This is a convenient system for us humans to understand, but unfortunately computers live in a world of their own where numbers with a base of ten do not mean an awful lot.

Inside a computer everything is stored as 'bits'. A bit is simply the smallest piece of information that a computer can handle, and a bit can either be turned on, or off. Another way to think of that would be to say that a bit can either equal one, or zero. In other words we have just two numeric symbols to work with, and thus computers are said to work with a binary system.

Representing numbers in binary is extremely tedious, as the number of zeroes and ones required to represent, say, 1938, is a very large number. In fact, it works out to be 11110010000, a ludicrous figure to deal with for humans, albeit a convenient one for computers.

So the inevitable happens and we have to arrive at a compromise.

There are a number of systems available which have been introduced over the years, but arguably the most popular, and certainly the easiest to grasp, is the hexadecimal one.

This has 16 numeric symbols to play with (hence the term hexadecimal), but since our own decimal system has a mere 10, we have to resort to using other things. The system in fact uses letters of the alphabet, and in full the symbols used are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Here the letter A in the hexadecimal system (let's call it hex from now on) is used to represent the number 10 in the decimal system, B represents 11, and so on up to F, which is used to represent 15.

So, our earlier number 1938 if it is expressed in hex becomes \$0790, the dollar sign telling us that this is a hex number. To check it, it is equal to 0 times 16 to the power zero, plus 9 times 16 to the power one, plus 7 times 16 to the power 2, or 0 plus 146 plus 1792, which is indeed equal to 1938.

Hex numbers can look a little confusing at first, but within a relatively small space of time you'll find them easy enough to use.

## **Bits and bytes**

How does this help us and the computer? It's all a question of bits and bytes. A bit, as we've seen, can be either on or off, and is the smallest amount of information that a computer can handle.

Being a small amount of information, everybody likes to speed the action up by handling more than just one bit at a time. On the Commodore 64 bits are grouped together in units of 8, and this grouping of 8 bits is called a byte. Incidentally, half a byte is called a nibble!

So inside one byte we have 8 bits. This can be looked at as follows:

Bit	0	1	2	3	4	5	6	7
Value	1	2	4	8	16	32	64	128

The values are arrived at using the binary system, and as you can see are simply multiples of 2. Now we've seen that a bit can be either on or off, so if every bit was turned on the value that would be held in that byte would be equal to all those numbers added together. You

probably won't be too surprised to learn that 128 plus 64 plus ... etc. is equal to 255, the maximum value that can be put (or POKEd) into any particular byte.

Now you can see why POKEing about inside the computer manages to produce such amazing results. We're not only affecting that memory location, we're also affecting every bit in it as well. Thus POKEing 1024 with, say, 77, turns on the bits that add together to give a value of 77, and turns off all the others. In this case bits 6, 3, 2 and 0 will be turned on, and bits 1, 4, 5 and 7 will be turned off.

If we want to alter bits 6, 3, 2 and 0, but keep the others as they were regardless of whether they are on or off, we need to use the OR command, in the form:

```
POKE LOCATION,PEEK(LOCATION) OR 77
```

This can be important in some cases, where we want various bits to remain as they were, but to alter some of the others for a particular purpose.

We'll be seeing a lot more of this later!

## **And now some graphics**

Now that we know all about hex and decimal, PEEK and POKE, bits and bytes, let's start on the first major section of this book, which deals with the powerful graphical capabilities of the Commodore 64.

## 2

# General Introduction to Graphics

The concept of using graphics on home computers is relatively recent, as indeed are home computers themselves. Only with the advent of machines like the Commodore 64 could one seriously talk about displaying in detail, sometimes incredible detail, any kind of graphical information.

Going back in time a little, the earlier Commodore computers all suffered when trying to display graphical characters. The Vic 20 had the rather obvious limitation of its 22 character wide screen. As mentioned in the introduction, the smallest item of information that the computer can handle is a bit. Graphically a bit is referred to as a pixel, which most people are inclined to think of as a small creature that lives at the bottom of the garden. However, in the world of graphics a pixel is the smallest thing that can be displayed on the screen.

## Screen resolution

The actual characters that you see on the television or monitor screen are themselves made up of several pixels grouped together on a grid eight pixels square. In chapter five we'll be giving you a program that allows you to redraw these characters and create your own character set (as well as showing you one intriguing POKE that instantly alters the 64 character set if used in conjunction with colours 8 to 15).

Thus the maximum screen resolution that can be achieved is directly linked to the number of characters across the screen that the computer is capable of displaying. Since each character is 8 pixels across, multiplying the number of characters by 8 gives you the maximum resolution. So in the case of the Vic 20 we're limited to 22 times 8, or 176 pixels: not very much.

Since the Commodore 64 displays a more acceptable 40 characters across the screen, this gives us a resolution of 40 times 8, or 320,

horizontally. As it also displays 25 columns on the screen, this in turn gives us a vertical resolution of 25 times 8, or 200 pixels.

Incidentally, you'll recall that the maximum number that you can POKE into any one register is 255. We've just mentioned that the horizontal resolution is 320, so how can you alter the last (320-255) 65 lines on the screen? In fact, you have to use two registers, as we'll see in chapter four.

But it is worth mentioning it here, as it explains a fact or two about another well known microcomputer, namely the Spectrum. Why has that got a screen display of 32 characters across? Well, take 32 times 8, which equals 256, and the maximum number you can POKE into a register is 255 (which is equal to 256 different values from 0 to 255), and so graphics on the Spectrum are a whole lot easier to handle, as we only have to worry about one register handling certain things, not two! Easier both for the programmer and for the computer.

## **Early days**

Before the Vic 20 Commodore had brought us the PET range of computers, with a screen display ranging from 40 to 80 characters across. However, the program to handle what would appear to be a very high resolution display was not written into those earlier machines, and had to be bought in from some other, independent supplier.

Now it's all there if we want it, but they've still made it difficult to get at! Oh well, we'll get there in the end.

Some of the things that we'll be covering later on include using the existing character set (all those funny little symbols on the front of the keys) to plot up histograms and bar charts, and to show just what can be done I've included a superb piece of animated graphic programming from one Don Denis. I take no credit for this: Don wrote it for the original PET machines many years ago, and I've just updated it to work on the 64.

After the existing character set we'll romp along into sprites, and give you a few sprite generators for single colour ones and multi-colour ones, as well as telling you how to handle many sprites on screen at the same time.

Following on from there we'll be showing you how to redefine characters, as well as giving you a program to make the whole job



an awful lot easier, and finally we'll end up with a look at high resolution plotting covering every pixel on the screen.

But before we get there, why is it necessary to tell you any of this at all? Shouldn't it be easy enough to do it all anyway?

### **With Commodore, nothing's easy!**

This statement is oh so true, as you'll probably have found out by now if you've attempted to cover anything graphical on the Commodore 64.

To define sprites, for instance, requires a rather large sheet of graph paper (or a pixel pad!), and some painstaking work with a set of coloured felt-tip pens, before performing some rather complicated mathematics and issuing a ridiculous string of POKE commands.

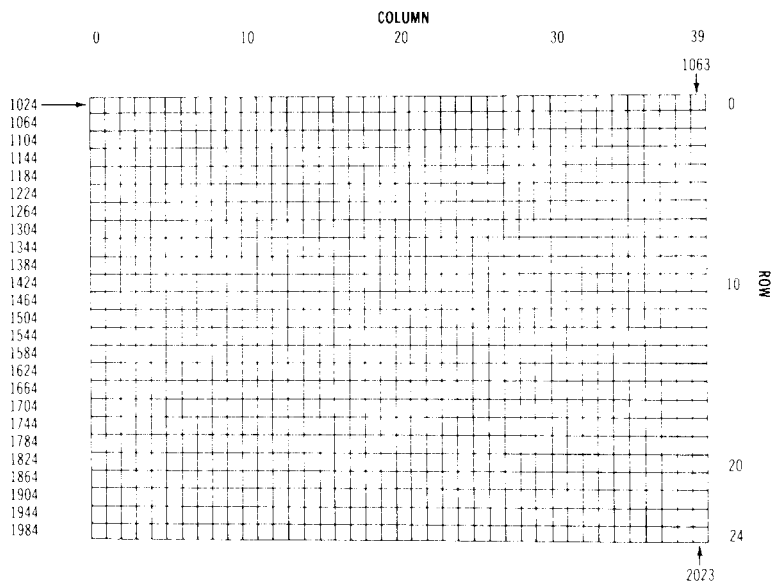
Don't worry: we'll make the going easier before we've finished.

There are no commands to help in ordinary everyday graphics either, unlike most of the other popular microcomputers around at the moment. The Electron and the ever-present Spectrum both come equipped with a fine set of commands to make life easier for us all, but Commodore chose to disregard everything and make life very difficult indeed.

True, once you do get a grasp of how everything works it can all become a whole lot easier, but getting started is certainly a struggle.

The simple PRINT command, which you might reasonably expect would help us along the way, is of no real help at all, since PRINTing is slow, and you're very limited in what you can actually print onto the screen. You can't print sprites, for instance, or print onto a high resolution screen, although you can (with a bit of juggling) manage to print out some user defined symbols.

Everything has to be handled by a whole series of POKes, and to do that we need to know where to POKE things to. At the very simplest level, we need to know first of all where the screen starts in memory, and although this is an old illustration it's well worth repeating here for the sake of completeness.

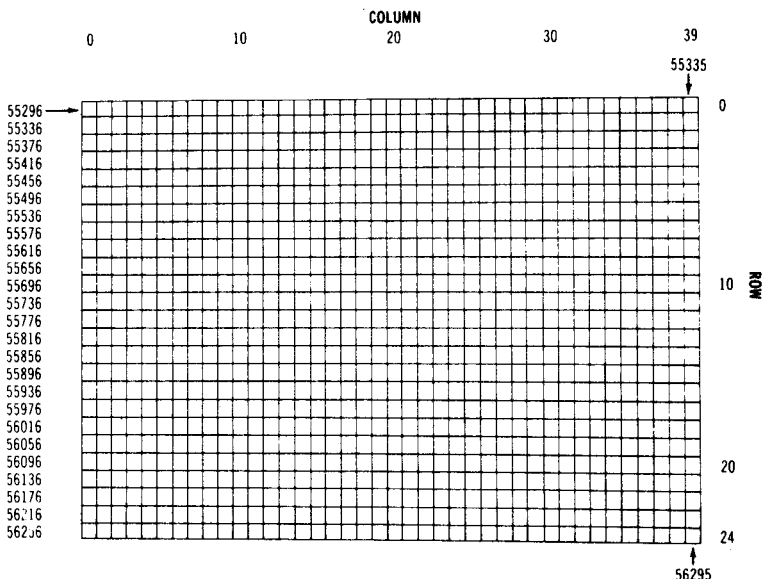


The top left-hand corner of the screen then is memory location 1024, and POKEing a few numbers into that location will soon display something on the screen.

As you can see, the screen covers 1,000 locations in all, and later on we'll be moving things about the screen with abandon. What, you mean you've tried POKEing things into location 1024 and nothing seems to have happened?

Well, we said it wasn't going to be easy! There is yet another memory map that you're going to need, and this one is concerned with the colour of objects as they appear on the screen.

Associated with every square on the screen is a corresponding memory location in the colour memory map. This starts off life at memory location 55296, and goes something like this:



## Altering colours on the screen

So in order to display something in any colour on the screen, we must not only POKE that something onto the screen in the first place, but we must also POKE a colour onto the same square.

Thus, in order to have, say, a black letter A in the top left hand corner of the screen we need to:

POKE 1024,65 : POKE 55296,0

The complete list of colours is as follows:

- |          |                  |
|----------|------------------|
| 0 BLACK  | 8 ORANGE         |
| 1 WHITE  | 9 BROWN          |
| 2 RED    | 10 LIGHT RED     |
| 3 CYAN   | 11 GREY COLOUR 1 |
| 4 PURPLE | 12 GREY COLOUR 2 |
| 5 GREEN  | 13 LIGHT GREEN   |
| 6 BLUE   | 14 LIGHT BLUE    |
| 7 YELLOW | 15 GREY COLOUR 3 |

These can also be accessed from the keyboard, by pressing the key marked CTRL and the key with the abbreviated colour word on it for the appropriate colour, which gives us the left hand side of the table listed above, or by pressing the key marked with the Commodore logo (known affectionately as the chicken head, from its strong resemblance thereto) and one of the colour keys for the right-hand side of the table.

To go back to the screen and colour locations for a while, keeping track of something on the screen requires us to alter continually two locations, starting at 1024 and 55296 respectively. A sensible idea when POKEing anything onto the screen is to declare two variables at the start of a program (say SS = 1024 and CS = 55296 for Screen Start and Colour Start respectively), and update both of them all the time, rather than trying to calculate new memory locations every time you want to move something. It's easier for you, the programmer, but it's also surprisingly quicker in execution time as well.

So, to bounce a little yellow blob up and down the screen, we need a program something like:

```
5 SS=1024:CS=55296
10 PRINT [CLR]
20 FORI=0TO24
30 POKE SS+I*40,81:POKECS+I*40,7
40 FORK=1TO50:NEXTK
50 POKE SS+I*40,32
60 NEXTI
70 FORI=24TO0STEP-1
80 POKE SS+I*40,81:POKECS+I*40,7
90 FORK=1TO50:NEXTK
100 POKE SS+I*40,32
110 NEXTI
120 GOTO 20
```

Explaining this line by line:

5 Declare variables for start of screen & colour

10 Clear the screen

20 Start of loop

30 Update SS and CS (81 is our little blob!)

40 Delay loop

20

50 POKE a space onto screen for animation effect

60 Next part of loop

70-110 Repeat everything in reverse order

120 Start the whole thing off again!

Not terribly exciting I grant you, but at least we've got something moving, in colour, on the screen! Altering the colour of our little blob by changing the values POKEd into CS in lines 30 and 80 will soon convince you which colours were meant to go together on the screen - and which were not!

Two more locations which are of interest while handling ordinary screen displays are memory locations 53280 and 53281.

These control the colour of the border and the background of your screen display respectively, and by altering them using the colours listed earlier you can get a combination of colours that you think looks best on the screen. You'll notice later that a number of the listings given prefer a black screen with mainly yellow text and a brown border. This produces a fairly restful image on the eye.

To see the whole lot in action:

```
10 PRINT [CLR]
20 FORI=0TO15
30 FORJ=0TO15
40 POKE 53280,J:POKE 53281,I
50 NEXTJ,I
```

You might like to insert a delay into this to slow the whole thing down a little, otherwise it looks a mite horrendous.

## A word of warning

Before we start on the lengthier listings, if you're going to attempt the laborious task of typing them in there are a few things you need to know before getting going.

These are concerned mainly with the idiosyncracies of the printer used for the listings in this book. First and foremost, it refuses to print any of the available Commodore 64 graphics keys, and where these have

been used, they've had to be replaced by an italicised version of the letter on which the graphic symbol is to be found.

For instance, the Alpine Slopes program requires the use of the shifted X character to represent something. In the listing, this has come out as an italicised X. We'll give greater warnings in the write-up for each listing, since some of the programs are designed to be run in lower case, and thus letters which are meant to appear in capitals have also come out in italics.

As if that wasn't enough, the listings have also had to be annotated, since the printer won't handle all the control codes used by the 64 to indicate cursor left, clear screen and so on, as well as changing colours, and the graphics symbols obtained with the Commodore logo key.

So here's a rundown of the symbols used to replace them: you've already seen one of them, [CLR] for clearing the screen. All you have to do is press the appropriate key on the keyboard.

[CR]	— Cursor Right	[CL]	— Cursor Left
[CU]	— Cursor Up	[CD]	— Cursor Down
[CLR]	— Clear Screen	[HOME]	— Cursor Home
[RVS]	— Reverse On	[OFF]	— Reverse Off

Colours are indicated by the three letter abbreviations on the front of the keys, so just press Control and 4 if the listing shows something like [PUR].

Where keys are to be repeated, this is indicated in the following fashion:

[CLR,2CD,5CR]

for instance, means 'clear the screen, press the cursor down key twice, and the cursor right key five times'.

Finally, the graphics keys accessed with the Commodore logo key (those on the left of the keys), are represented as [CBMC], or whatever, which means press the logo key and the C key at the same time.

## Conclusion

Using graphics on the Commodore 64 is not easy. Commodore seem to have gone out of their way deliberately to make life difficult for all

of us, by not including any graphics commands in the Basic repertoire supplied with the machine.

The 64 is capable of producing some very sophisticated displays indeed, as we will see, but to produce these requires a good deal of effort on the part of the programmer.

To work all the time in Basic is possible, although some of the results thus achieved will take quite some time to actually appear on the screen, and so a certain knowledge of machine code is desirable on the part of the programmer. This we'll be covering throughout the book, and in particular in chapters ten and eleven, where we'll be going deeply into machine code and assembling/disassembling of programs.

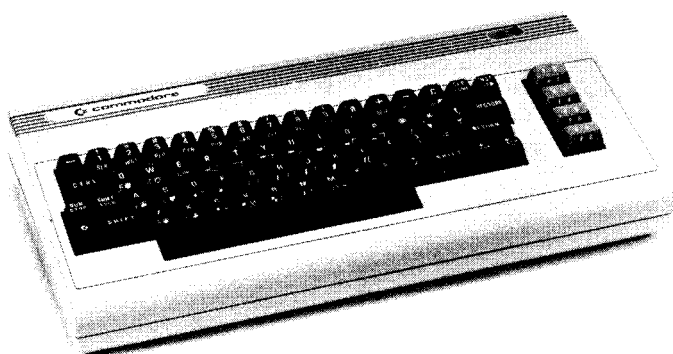
The next four chapters will introduce you to some of the things that can be achieved using the existing graphics character set, manipulating single and multi-colour sprites, producing and using user-defined characters, and finally covering the use of high resolution plotting on the screen.

Throughout each chapter there'll be a number of short example programs that you can type in just to see what they do, and also in each chapter there'll be a number of lengthier listings of such things as useful utilities for producing multi-coloured sprites or designing your own character set.

Finally, most of the chapters will have a 'non-utility' listing so that you can see what we've been talking about actually in action. The majority of these listings are just simple games, illustrating some of the concepts mentioned earlier on in the chapter. Each listing will have a full explanation of how it is doing what it is doing, and you are, of course, welcome to experiment with and expand on the listings as given.

Throughout the rest of the book it would probably be useful to have some kind of reference guide with you that gives details of machine code instruction sets, and other more technical data, since we're not going to fill pages and pages with extremely long reference charts. However, they are useful things to know, and some of the otherwise unexplained POKEs and machine code instructions might lead you astray.

Still, we'll try to explain everything as we get to it, so without further ado, let's take a look at the existing graphics set, and what you can do with it (in the nicest possible way, of course!).





# 3

## Using the Existing Graphics Set

### Introduction

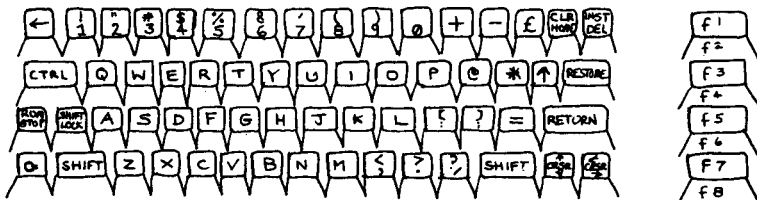
Before we can talk about using the existing character set, we need to know where everything is on the keyboard. This is not as fatuous a remark as it may sound, since some of the characters available to us are only accessible under certain circumstances.

Could you, for instance, immediately produce the tick mark on the screen? And before you start looking - no, it isn't marked on the keyboard!

The tick mark is, in fact, a mere POKE away. POKEing somewhere on the screen with a 122 brings the blighter up there, as long as you give it a colour as well.

But before introducing a strange set of POKes to you, let's stick with the Basic keyboard for a while and go on a tour.

### The Commodore 64 keyboard



You should by now be fairly familiar with the Return, Control and logo keys, as well as all the cursor movement ones. The function keys we'll leave out of it for now (we'll see them later in chapter ten), except to say that they can be used most simply by detecting their presence from within a program.

Keys can be detected in a variety of ways, but the simplest is probably the old stand-by:

```
10 GETA$: IFA$="" THEN10
```

In other words, if no key is being pressed, then loop back to line 10 again and wait until something is pressed. Alternatively, you can use the fact that PEEK(197) or PEEK(203) both return a value which relates to key being pressed, giving you 64 if nothing is currently being pressed.

So either way can be used to watch for various keys being used, and the function keys can be used in the first instance by checking for CHR\$ numbers 133 to 140, as in:

```
10 GETA$: IFA$<>CHR$(133) THEN10
```

which will sit there until you press function key 1. The other method could be written something like:

```
10 IFPEEK(197)<>64 THEN10
```

which again will sit and wait until you press key F1. To find out what the value stored in PEEK(197) is for every key on the keyboard, a simple program will allow you to check for each one as pressed:

```
10 IFPEEK(197)=64 THEN10  
20 PRINT PEEK(197):GOTO 10
```

Thus a whole host of keys could be checked for in a 'menu' situation, and the program written to react accordingly.

The majority of the other keys available from the keyboard that we haven't yet mentioned are the set of graphics characters, which we'll come to in a moment. The most interesting keys though are the ones labelled BLK, WHt, and so on.

## And on to colour

The use of colour is probably one of the greatest attributes of the Commodore 64, although as we'll see later the number of colours available to us at any one time is very much dependent on what graphic mode we happen to be in. The usual rule is that if we go for greater resolution on the screen, we can use increasingly fewer colours, and the opposite is also true: if we want to use more colours we must do so at the cost of the available resolution.

For example, ordinary sprites can be shown in just one colour, and in any position on the screen will appear to be in that colour, with the screen background colour showing through.

However, a multi-colour sprite can have three different colours used in it, as well as being able to show the screen background colour. However, a multi-colour sprite is more 'chunky' than its ordinary counterpart, since we have effectively to halve the horizontal resolution.

But on power-up we have immediate access to all the 16 available colours, and these can be displayed in a number of ways. The simplest is by using the control and logo keys in combination with an appropriate colour key. The following program allows you to check through all sixteen colours, and is, incidentally, a good check to see whether or not you've tuned in your television set correctly.

```
5 SS=1024:CS=55296
10 FORI=0TO15
20 FORK=0TO39
30 POKE SS+K+I*40,160:POKE CS+K+I*40,I
40 NEXTK
50 NEXTI
```



















When run, this program will produce a set of coloured bars across the screen. Using variations on a theme, a random checked pattern can be produced as follows:

```
10 A$="CTRL1,CTRL2,CTRL3, .... LOGO6,LOGO7,LOGO8"
20 PRINTMID$(A$,INT(RND(.5)*15+1),1)"[RVS] "I:GOTO
20
```









which defines the string A\$ to contain all the colours available (don't

type in the commas when you run the program!), and then picks out one of them at random and prints it out followed by a reverse space so you can see the colour. It then loops back and repeats the process ad infinitum until you press the STOP key.

Some of these colours (though not all) are also available using the CHR\$ command, as the following table shows:

PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$
	0		17	"	34	3	51
	1		18	#	35	4	52
	2		19	\$	36	5	53
	3		20	%	37	6	54
	4		21	&	38	7	55
	5		22	.	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES  	8		25	*	42	;	59
ENABLES  	9		26	+	43		60
	10		27	,	44	=	61
	11		28	-	45		62
	12		29	.	46	?	63
	13		30	/	47	@	64
	14		31	0	48	A	65
	15		32	1	49	B	66
	16	!	33	2	50	C	67

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
D	68		97		126		155
E	69		98		127		156
F	70		99		128		157
G	71		100		129		158
H	72		101		130		159
I	73		102		131		160
J	74		103		132		161
K	75		104	f1	133		162
L	76		105	f3	134		163
M	77		106	f5	135		164
N	78		107	f7	136		165
O	79		108	f2	137		166
P	80		109	f4	138		167
Q	81		110	f6	139		168
R	82		111	f8	140		169
S	83		112		141		170
T	84		113		142		171
U	85		114		143		172
V	86		115		144		173
W	87		116		145		174
X	88		117		146		175
Y	89		118		147		176
Z	90		119		148		177
[	91		120		149		178
£	92		121		150		179
]	93		122		151		180
↑	94		123		152		181
←	95		124		153		182
	96		125		154		183

PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$
	184		186		188		190
	185		187		189		191

<b>CODES</b>	<b>192-223</b>	<b>SAME AS</b>	<b>96-127</b>
<b>CODES</b>	<b>224-254</b>	<b>SAME AS</b>	<b>160-190</b>
<b>CODE</b>	<b>255</b>	<b>SAME AS</b>	<b>126</b>

From the above table you can see that all of the graphics characters are also available by a series of CHR\$ commands. However, attempting to display them all using CHR\$ will result in strange things happening on the screen, as some of these commands are used for things other than displaying characters. CHR\$(13) for instance, is the equivalent of pressing the Return key, CHR\$(147) is the same as clearing the screen, and so to display all the characters on the screen we need to resort to our old friend POKE, as in the following example:

```

5 PRINT "[CLR]":SS=1024:CS=55296
10 POKE 53280,9:POKE 53281,0
20 FORI=0TO255
30 POKESS+I,I:POKE CS+I,7
40 NEXT I

```

This simply clears the screen, sets the border and background colours to be brown and black, and then POKes every character onto the screen in yellow for legibility.

The program can be made a lot more interesting by incorporating an instruction to POKE the computer repeatedly into graphics and lower case. As you know, this can be achieved from the keyboard by pressing one of the shift keys and the logo key at the same time. But from within a program we have to use yet another POKE command, like this:

```

5 PRINT "[CLR]":SS=1024:CS=55296
10 POKE 53280,9:POKE 53281,0
20 FORI=0TO255
30 POKESS+I,I:POKE CS+I,7
40 NEXT I
50 POKE 53272,23
60 FORK=0TO500:NEXTK
70 POKE 53272,21
80 FORK=0TO500:NEXTK
90 GOTO 50





































































```

The additional lines now perform a continuous loop, with the FOR K ... lines acting purely as a delay to allow you to see what is happening. The two POKES in lines 50 and 70 respectively put us into lower case and graphics modes.

If we wanted this to be of more practical use, so that we could tell which value of I was producing which particular character on the screen, we'd have to alter line 30 to POKE the characters out on, say, every third square, and in the gap in between use the PRINT command to put the value of I that belongs to that character on the screen.

However, if you want it in reference form, here it is:

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	S	s	19	&		38
A	a	1	T	t	20	'		39
B	b	2	U	u	21	(		40
C	c	3	V	v	22	)		41
D	d	4	W	w	23	*		42
E	e	5	X	x	24	+		43
F	f	6	Y	y	25	,		44
G	g	7	Z	z	26	-		45
H	h	8	[		27	.		46
I	i	9	£		28	/		47
J	j	10	]		29	0		48
K	k	11	↑		30	1		49
L	l	12	←		31	2		50
M	m	13	SPACE		32	3		51
N	n	14	!		33	4		52
O	o	15	"		34	5		53
P	p	16	#		35	6		54
Q	q	17	\$		36	7		55
R	r	18	%		37	8		56

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
9		57		Q	81			105
:		58		R	82			106
:		59		S	83			107
<		60		T	84			108
=		61		U	85			109
>		62		V	86			110
?		63		W	87			111
		64		X	88			112
	A	65		Y	89			113
	B	66		Z	90			114
	C	67			91			115
	D	68			92			116
	E	69			93			117
	F	70			94			118
	G	71			95			119
	H	72	SPACE		96			120
	I	73			97			121
	J	74			98			122
	K	75			99			123
	L	76			100			124
	M	77			101			125
	N	78			102			126
	O	79			103			127
	P	80			104			

Codes from 128-255 are reversed images of codes 0-127.



## Tricks from the keyboard

Having got all our characters up on the screen, and achieved an overall familiarity with the keyboard (what characters appear when you POKE them to the screen, and so on), it only remains to give you a few tricks of the trade before we get down to some serious programming.

While not specifically connected to either graphics or sound, these tricks do have their uses when programming generally, giving you such things as repeat keys, preventing people from looking at your program listings, disabling run/stop and restore, and so on.

In order, we have:

COMMAND	USE	REMEDY
POKE 775,200	PREVENT LISTING	POKE 775,167
POKE 808,239	DISABLE STOP KEY	POKE 808,237
POKE 808,255	AND RESTORE KEY!	POKE 808,237
POKE 649,0	DISABLE KEYBOARD	POKE 649,10
POKE 808,255	PREVENT SAVING	POKE 808,237
POKE 818,32	AND LISTING	POKE 818,237
POKE 650,255	AUTO REPEAT	POKE 650,0

If you discover any other 'peculiarities' of the machine, please write and let us know!

## Graph plotting

Obviously with all the graphical features we have at our disposal on the Commodore 64, there are many uses to which even the existing character set can be put. After all, it didn't deter the earlier Commodore computers from producing some excellent results.

Of particular interest in the world of animation would be the characters to be found on the U,I,J and K keys by using the shift key. Little aliens in flight could easily be achieved by using these characters and

swopping from one to another.

For graph plotting, we could use the quarter square graphics characters found on the D,F,C and V keys, as this would relatively easily double the resolution on the ordinary 64 from the 40 by 25 characters achieved at power on to a more respectable 80 by 50 by careful use of those four characters.

Bar charts, histograms and pie charts are all fairly easily done, as are simulations of mathematical equations in graphical form, for instance the plotting of sine waves, or two equations combined together to give one output.

For example, if plotting a bar chart you could use a string that was set up to be four reverse spaces in length, with the first two being light red, and the second two being dark red. A little juggling with reverse field characters thereafter gives a very nice three-dimensional effect on the screen. You could also use the other two colours that have light and dark versions, namely green and blue, and perhaps also black with one of the darker grey colours.

Experiment is the key here, and if you manage to produce a nice bar chart and someone says 'What about histograms?', simply turn the television set on its side and there you are: instant histograms!

Some very good effects can be achieved by this use of reverse field characters: some of them if printed out in reverse look surprisingly different from their more normal form.

## **Android Nim**

Use of reverse field characters is made to devastating effect in the following program from Don Denis. In it he defines a number of little androids made up out of reverse quarter block (and other!) characters, and then proceeds to bring in some delightful animation as the game proceeds.

The program itself is based on the old game of Nim, usually played with three rows of matches, with three matches in the first row, five in the second and seven in the third.

Players take it in turns to remove matches, and whoever takes the last match wins. So the trick is to pick and choose both the row and the number of matches to remove, and force your opponent into a

position where he has to give you the last one.

Here, the game is acted out by eighteen androids, three acting as executioners, and the other fifteen forming our three rows of matches. You and the computer alternately select the row and the number of androids to be 'executed', and try to force your opponent into giving you the last 'droid. The game shows a cleverly increasing I.Q. if it loses the first few games, and becomes gradually more and more difficult to beat.

The listing itself is not without pitfalls when entering it in. In common with all the listings in this book we've removed all the cursor control characters, graphics characters and so on, and replaced them with the symbols given earlier.

With this one more than any of the other listings you're going to have to take great care when entering it, especially lines 41 to 47 and the data statements in lines 5030 to 5120.

The data statements in particular need very careful attention. Since we've replaced the graphics that would normally appear with their worded equivalent, some of the data statements are actually longer than one line (e.g. line 5030), and we've had to run them on over one or two lines. When you type it in, make sure that all the data just comes in one data statement. In other words, where the REM statement says 'follows on from lines XXXX and YYYY), the group of lines before it are intended to be one line of data.

Finally, the pound signs really are meant to be pound signs, and the occasional letter in *italics* is meant to be in upper case when you type it in. Owing to the POKE 53272,23 statement in line 0, the game is meant to be played in upper/lower case rather than upper case/graphics mode, so just type everything else in as normal. When run, you'll get the correct mixture of upper and lower case letters.

Simply sitting back and watching the little characters can be enjoyable enough, although you might get tired of their remarks as you continually do nothing. When the game is being played properly, watch out for their reactions as they are about to shoot their fellow androids, or if you set them an impossible task or type in something they don't understand.

This program has been a classic ever since it first appeared on a Commodore PET in 1979. I've simply converted it to run on the 64 and given each of the three main androids a different voice.

A very clever use of the existing graphics set.

```
0 POKE 53280,1:POKE 53281,1:POKE 53272,23
1 PRINT"[CLR]"TAB(10)"[2CD,GRN]***[RVS]ANDROID #IM
[OFF]***"
2 PRINTTAB(18)"[CD]BY":PRINTTAB(14)"[CD]DON DENISE
3CD]"
3 PRINTTAB(11)"TORONTO, CANADA":PRINTTAB(13)"JULY,
1979":
4 REM 153 UNDERHILL DR
5 REM DON MILLS, CANADA
6 REM M3A 2K6
7 REM (416)445-3927
8 PRINTTAB(12)"[3CD]AMENDED BY"
9 PRINTTAB(12)"[CD]PETE GERRARD":PRINTTAB(13)"JAN.
, 1984"
10 SF=64:VC=54272
12 POKEVC+24,15
14 POKE VC+19,18:POKE VC+20,250
18 POKE VC+18,33
31 CL$="[HOME,40SP,HOME]"
34 LN=214:CN=211:KB=198
35 DEF FNE(X)=(A(P)ORE)AND(NOT(A(P)ANDE)):IQ=.7
36 DIM B$(18)
38 : FORI=0TO17
39 : READB$(I)
40 : NEXTI
41 B$(18)="[CL,RVS,SP,OFF,2CBMK,RVS,2SP,CD,SCL,SP,
CBMC,OFF,CBMD,SP,RVS,SP,CD]"
42 B$(18)=B$(18)+"[SCL,SP,CBMC,OFF,CBMV,CBMD,RVS,S
P,CD,SCL,SP,OFF,CBMK,SP]"
43 B$(18)=B$(18)+"[CBMD,RVS,CBMF,CD,SCL,SP,OFF,CBM
K,SP,RVS,2CBMK,CD,SCL]"
44 B$(18)=B$(18)+"[SP,CBMV,CBMK,OFF,CBMD,RVS,SP,CD
,SCL,2SP,2CBMK,SP,CD,SCL,SP]"
45 B$(18)=B$(18)+"[CBMC,CBMV,OFF,CBMI,RVS,SP,SCL,7
CU,OFF,5SP,CD,SCL,5SP]"
46 B$(18)=B$(18)+"[CD,SCL,5SP,CD,SCL,5SP,CD,SCL,5S
P,CD,SCL,5SP,CD,SCL,5SP]"
47 B$(18)=B$(18)+"[CD,SCL,5SP,CU]"
50 DIM PX(17),PY(17),R(17),CM$(7),A(2),B(2)
60 FORI=0TO17
70 : READ PX(I),PY(I)
75 : R(I)=I
80 : NEXTI
105 DIM M$(15)
110 FORI=0TO15
115 : READ M$(I)
120 : NEXTI
```

```

121 FORI=0TO7
122 : READCM$(I)
123 : NEXTI
130 GOSUB2000
146 IQ=.9
150 RR=3:B(0)=10:B(1)=15:B(2)=18
155 Q$="DO YOU NEED INSTRUCTIONS?":GOSUB800
160 IFA$="N"GOTO200
165 Q$="WE ARE THE EXECUTIONERS.£ PICK ONE OF US (
A B OR C)£ TO DESTROY AS MAN
166 Q$=Q$+"Y ANDROIDSE£ FROM EACH ROW AS YOU WISH.£
THEN IT IS OUR TURN TO PLAY.
167 Q$=Q$+"£ THE ONE WHO GETS THE LAST DROID WINS.
":GOSUB1500
200 PRINT"[CLR]":GOSUB2000:FOR N=3TO17
205 : GOSUB1000
210 : R(N)=N
215 : NEXTN
220 RR=18:A(0)=7:A(1)=5:A(2)=3
225 TR=0:Q$="DO YOU WANT TO PLAY FIRST?":GOSUB800
228 M=0
230 IFA$="N"GOTO245
235 IFA$<>"Y"GOTO225
240 M=1-M
245 IFRR=3GOTO500
250 IFM=0GOTO400
255 TR=0:Q$="IT IS YOUR TURN.£ WHICH ROW?":GOSUB800
0
256 Z=1
260 P=ASC(A$)-65
265 IFP<0ORP>2THENGOSUB600:GOTO255
270 IFA(P)=0THENGOSUB650:GOTO255
275 TR=P:Q$="HOW MANY ANDROIDS?":GOSUB800
280 Z=ASC(A$)-48
285 IFZ<1ORZ>9THENGOSUB600:GOTO255
288 POKELN,PY(P):POKECN,PX(P):PRINT"[2CU,CR]Z
290 IFZ>A(P)THENGOSUB650:POKELN,PY(P):POKECN,PX(P)
:PRINT"[2CU,2CR,SP]":GOTO275
300 SL=25:GOSUB700
305 POKELN,PY(P):POKECN,PX(P):PRINT"[2CU,2CR,SP]"
310 GOTO240
400 E=0:F=0
405 FORP=0TO2
410 : E=FNE(0):IFA(P)>FTHENF=A(P):I1=P
415 : NEXTP
420 FORP=0TO2
425 : R=FNE(0):IFR<=A(P)GOTO470
430 : NEXTP:STOP
470 IFR=A(P)ORIQ>RND(1)THENP=I1:R=A(P)-INT(RND(1)*
(A(P)-1)+1)
475 TR=P:Z=A(P)-R:Q$="WE CHOOSE"+STR$(Z)+" ANDROID
FROM ROW "+CHR$(P+65)+"£"

```

```

476 GOSUB1500
478 SL=5:GOSUB700
495 GOTO240
500 Q$=" WIN.£": IFM<>0THENQ$=" LOSE.£"
505 Q$="YOU"+Q$
510 IFM=0THENQ$=Q$+" WE WILL PLAY BETTER NEXT TIME
.£": IQ=IQ*IQ*IQ
515 TR=0:GOSUB1500
520 Q$="WOULD YOU LIKE ANOTHER GAME?":GOSUB800
525 IFA$<>"N"GOTO200
530 Q$="THANK YOU FOR PLAYING.££":GOSUB1500:RUN
600 TR=0:R1=0:R2=0:R3=0:SL=17
605 M1$=M$(9):M2$=M$(10):M3$=M$(11)
610 GOSUB900
615 Q$="YOUR ANSWER DOES NOT MAKE SENSE.£"
616 IFZ=0THENQ$="CAN'T YOU MAKE UP YOUR MIND?£"
617 GOSUB1500
620 RETURN
650 R1=P:R2=P:R3=P:SL=25
655 M1$=M$(7):M2$=M$(8):M3$=M$(8)
660 GOSUB900
665 TR=P:Q$="SORRY, ONLY"+STR$(A(P))+ " ANDROIDS LE
FT.£"
670 IFA(P)=0THENQ$="I CAN'T DO IT. I HAVE NONE LEF
T.£"
675 GOSUB1500
680 RETURN
700 R1=P:R2=P:R3=P
705 M1$=M$(6):M2$=M$(8):M3$=M$(8)
710 GOSUB900
712 II=B(P)-A(P)
715 FORI=IIT0II+Z-1
720 : POKELN,PY(I):POKECN,PX(I):PRINT"[CU,CR]"B$(6
)
725 : NEXTI
726 REM
727 FORJJ=255TO30STEP-1:POKE VC+15,JJ:POKE VC+14,J
J:NEXTJJ
728 POKE VC+15,0:POKE VC+14,0
730 FORI=1TOZ
735 : GOSUB950
740 : NEXTI
788 RETURN
800 POKEKB,0:QU$=Q$:GOSUB1500
805 T=TI+800
810 M1$=M$(RND(1)*16)
815 M2$=M$(RND(1)*16)
820 M3$=M$(RND(1)*16)
825 R1=R(RND(1)*RR)
830 R2=R(RND(1)*RR):IFR2=R1GOTO830
835 R3=R(RND(1)*RR):IFR3=R2ORR3=R1GOTO835
840 SL=INT(25*RND(1)+1)

```

```

845 GOSUB900
850 GETA$: IFA$<>" " THEN PRINT CL$: RETURN
855 IF TI>T THEN Q$=CM$(RND(1)*8)+ "£ " + QU$: GOSUB1500
: GOTO805
860 GOTO810
900 FORC=SL TO1STEP-1
910 : POKELN,PY(R1):POKECN,PX(R1):PRINT"[CU,CR]"B$
(ASC(RIGHT$(M1$,C))-SF)
920 : POKELN,PY(R2):POKECN,PX(R2):PRINT"[CU,CR]"B$
(ASC(RIGHT$(M2$,C))-SF)
930 : POKELN,PY(R3):POKECN,PX(R3):PRINT"[CU,CR]"B$
(ASC(RIGHT$(M3$,C))-SF)
940 : NEXTC
945 RETURN
950 POKELN,PY(R1):POKECN,PX(R1):PRINT"[CU,CD,4CR]"
;
954 FORJJ=20TO140STEP7:POKEVC+15,JJ:POKE VC+14,JJ:
NEXTJJ:POKEVC+15,0:POKEVC+14,0
955 SP=PX(R1):EP=PX(B(P)-A(P))-5
959 SP=PX(R1):EP=PX(B(P)-A(P))-5
960 FORJ=SPTOEPSTEP2:PRINT" -=*[3CL]";:NEXTJ
965 IFINT((EP-SP)/2)*2=EP-SP THEN PRINT"[CL]";
970 PRINT"[CU,CR]"B$(18)
974 RR=RR-1:A(P)=A(P)-1
976 A=3
977 ONP+1GOTO990,985,980
980 A=A+A(1)
985 A=A+A(0)
990 FORJ=ATO16
991 : R(J)=R(J+1)
992 : NEXTJ
998 RETURN
1000 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(1+
7*RND(1));
1010 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(0)
:
1020 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(9+
5*RND(1));
1030 POKELN,PY(N):POKECN,PX(N):PRINT"[CU,CR]"B$(14
+4*RND(1));
1040 RETURN
1500 PRINTCL$
1505 II=0:GOSUB1600
1510 FORI=1TOLEN(Q$)
1515 : CH$=MID$(Q$,I,1)
1517 N=N+1
1520 : IFCH$=" " THENGOSUB1600
1525 : IFCH$="£" THENII=I:FORJ=1TO600:NEXTJ:PRINTCL
$:GOTO1550
1530 : POKELN,1:POKECN,I-II:PRINT"[CU,CL]"CH$
1550 : NEXTI
1560 RETURN

```

```

1600 POKELN,PY(TR):POKECN,PX(TR):PRINT"[CU,CR]"B$(
1);
1605 POKE VC+15,30-TR*10:POKE VC+14,10
1606 POKE VC+15,0:POKE VC+14,0
1610 PRINT"[3CL,CBMK,CL]":GOSUB1700:REM USE CBM K
EY AND K FOR GRAPHIC CHARACTER
1615 PRINT"[RVS,CBMC,CL]":GOSUB1700:REM USE CBM K
EY AND C FOR GRAPHIC CHARACTER
1620 PRINT"[SP,CL]":GOSUB1700
1625 PRINT"[CBMC]":GOSUB1700:REM USE CBM KEY AND C
FOR GRAPHIC CHARACTER
1630 N=0
1650 RETURN
1700 FORJJ=1TO3*RND(1)
1702 POKE VC+15,30-(TR*10+RND(1)*3)
1704 POKE VC+14,30-(TR*10+RND(1)*3)
1706 NEXTJJ
1710 POKE VC+15,0:POKE VC+14,0:RETURN
2000 FOR N=0TO2
2010 : GOSUB1000
2020 : PRINT"[RVS,2CU,3CL]*[CD,CL]"CHR$(N+65)
2030 : NEXTN:RETURN
5030 DATA"[RED,3CD,CR,RVS,SP,CD,CL,SP,CU,CBMD,OFF,
CD,CL,CBMK,CD,2CL,2CBMK,CD
5031 DATA"[2CL,2CBMK,CD,3CL,CBMC,CBMV,RVS,CBML,OFF
,3CL,7CU,RED]"
5032 REM CARRY ON FROM LINES 5031 AND 5030
5035 DATA"[RED,SP,2CBMK,CD,3CL,RVS,CBMV,2CBMF,OFF,
CBMF,CD,4CL,CBMC,RVS,CBMC
5036 DATA"[CBMD,OFF,SP,RED]":REM CARRIES ON FROM L
INE 5035
5040 DATA"[BLU,SP,2CBMD,CD,3CL,RVS,CBMK,CBMB,CBMV,
OFF,SP,CD,4CL,RVS,CBML,SP
5041 DATA "[CBMD,OFF,CBMV,RED]":REM CARRY ON FROM
LINE 5040
5045 DATA"[PUR,SP,2CBMD,CD,3CL,RVS,CBMV,2SP,OFF,CB
MF,CD,4CL,RVS,CBMK,CBMB,CBMV
5046 DATA"[OFF,SP,RED]":REM THIS CARRIES ON FROM L
INE 5045
5050 DATA"[PUR,SP,RVS,2CBMK,CD,3CL,CBMK,2CBMD,OFF,
SP,CD,4CL,OFF,CBMC,RVS,SP
5051 DATA"[CBMB,OFF,SP,RED]":REM THIS CARRIES ON F
ROM LINE 5050
5055 DATA"[BLK,2SP,CBMK,CD,3CL,RVS,CBMK,SP,OFF,CBM
V,SP,CD,4CL,CBMC,RVS,SP,CBMD
5056 DATA"[OFF,SP,RED]":REM WHEN YOU TYPE IT IN, C
ONT. THIS AS PART OF LINE 5055
5060 DATA"[RVS,2CBMK,OFF,SP,CD,3CL,CBMC,RVS,CBMD,S
P,OFF,SP,CD,4CL,CBMC,RVS,CBMV,
5061 DATA"[CBMD,OFF,SP]":REM WHEN YOU TYPE IT IN,
CONT. THIS AS PART OF LINE 5060
5065 DATA"[SP,CBMK,2SP,CD,4CL,SP,RVS,CBMF,SP,OFF,S

```



P,CD,4CL,CBMC,RVS,SP,CBMD,OFF,SP]  
 5070 DATA"[CD,CR,RVS,2SP,OFF,CR,CD]"  
 5075 DATA"[3CD,RVS,CBMB,CD,2CL,OFF,CBMC,CBMF,CD,2CL,SP,CBMC]"  
 5080 DATA"[BRN,3CD,RVS,CBMB,CD,2CL,OFF,SP,CBMK,CD,2CL,SP,CBMC,RED]"  
 5085 DATA"[BRN,3CD,RVS,CBMB,CD,2CL,OFF,SP,CBMK,CD,2CL,SP,CBMV,RED]"  
 5090 DATA"[GREY1,3CD,RVS,CBMB,CD,2CL,OFF,SP,CBMK,CD,2CL,CBMC,SP,RED]"  
 5100 DATA"[LT.RED,3CD,RVS,CBMK,CD,CL,CBMK,2CL,OFF,SP,CD,CL,SP,CBMV,RED]"  
 5105 DATA"[YEL,3CD,3CR,CBMF,CD,CL,RVS,CBMB,CD,2CL,CBMD,OFF,SP,RED]"  
 5110 DATA"[YEL,3CD,3CR,CBMF,CD,CBMK,CD,2CL,RVS,CBMD,OFF,SP,RED]"  
 5115 DATA"[GRN,3CD,3CR,CBMF,CD,CL,CBMK,CD,2CL,CBMK,CBMV,RED]"  
 5120 DATA"[GRN,3CD,3CR,CBMF,CD,CL,CBMK,CD,2CL,CBMK,CBMC,RED]"  
 5230 DATA0,2,3,10,0,18,5,2,10,2,15,2,20,2,25,2,30,2,35,2,13,10,18,10,23,10,28  
 5240 DATA10,33,10,21,18,26,18,31,18  
 5330 DATA AHDEEDABACABACABACAADHDAB  
 5335 DATA AHDAFADAFADHDHAFFHFFFAA  
 5340 DATA AHANCAAAABKPLQAKPINHACCAF  
 5345 DATA JOKPLQKPJOKPLQKPJOINFJHFM  
 5350 DATA FGKMLJLJLJLJLFHFFADEQNJNID  
 5355 DATA AHAFADAFAHADDFDFDHDFAFGKN  
 5360 DATA AHBBBAHADEEEDABACABACADEI  
 5365 DATA ABBBAHADEEEDAFADAFADAFABA  
 5370 DATA OJJJPPPPQPQPKKKKKKKKKKKKK  
 5137 DATA AAAAAAAAAHBBBAAACCAHAAAAHA  
 5380 DATA AAAAAAAAAIIIIJKLLLLIIIIII  
 5385 DATA AAAAAAAAAIIIIOPQQNNNNNNNN  
 5390 DATA AHABADACAFABADACFBDCFBDHD  
 5395 DATA ADEDADEDAEDHAFGFAGFAFGF  
 5400 DATA BDBDBDBDBACFMNCACACACAHCA  
 5405 DATA AFGGNQPPQFAHDEPQNDAFGLIG  
 5510 DATA"COME ON.", "WE HAVEN'T GOT ALL DAY!"  
 5520 DATA"WE HAVE BETTER THINGS TO DO."  
 5530 DATA"JUST ANSWER THE QUESTION.", "IT ISN'T THAT DIFFICULT!"  
 5540 DATA"THERE IS A LIMIT TO OUR PATIENCE!"  
 5545 DATA"JUST GET ON WITH IT.", "WHY ARE YOU SO STUPID!?"

READY.



## 4

# Sprites

### Just what is a sprite?

A sprite sounds as if it ought to be some kind of elf dancing about through the woods, but reality and computing have a way of bringing you back down to earth, and we find that a sprite is in fact a programmable, moveable object block that can either be 24 pixels horizontally by 21 pixels vertically, or in some special cases as we'll see in chapter five, 24 pixels by 24.

For now we'll confine ourselves to standard sprites as they exist on the Commodore 64.

24 pixels by 21 is approximately three characters wide by three characters high, and when in normal mode this is indeed about the size of a sprite. However, as we shall be seeing later, sprites can be expanded in either the horizontal or the vertical direction (or indeed both), to roughly twice this size.

As we've already seen, a pixel can either be on or off, and so an ordinary sprite like this can have every one of the 24 by 21 pixels defined to be either on or off. In other words, the image portrayed by the sprite will be a two colour one: one colour is the actual sprite colour, and the other will be whatever background colour the screen happens to be at the time.

Commands exist to move sprites about the screen; you can check whether a sprite has hit another sprite (or some other data on the screen), and they can be turned on or off selectively. That is, we can have sprites 0,1 and 3 turned on, and sprite 2 turned off, should we so desire it.

### Data storage

Needless to say, there are one or two penalties associated with using

sprites, and one of them is that we've got to store the data for each sprite somewhere in the computer's memory.

Since 24 by 21 pixels is equal to 63 bytes, defining a large number of sprites can take up a lot of memory. However, a practical limit for most purposes will be eight sprites, and to begin with we'll just stick with one or two.

There are eight locations in memory which tell the computer where the data is to be stored for each of the first eight sprites. These are locations 2040 through to 2047, which are the eight locations immediately before the start of the computer's Basic RAM memory.

If we POKE a 13 into location 2040, this tells us and the computer that the 63 bytes of data for sprite 0 are to be found in the 13th location of the computer's memory set aside for sprite storage. This starts at memory location 832. Why 832? Well, you're probably getting used to the fact that in the world of computers, everything has inevitably got to be divisible by 8, and 63 is not particularly useful as a number to be divided by 8. So the computer sets aside 64 bytes for each sprite, calling the last one a 'sprite marker'. 64 times 13 equals - you guessed it - 832, and so that's where the 13th block of sprite data lives.

The 14th and 15th block are the next two 64 byte chunks of memory, starting at locations 896 and 960 respectively. The 12th block is also used by the computer itself, and so for another one we must move down in memory to the 11th block, starting at location 704 (11 times 64).

So if we'd POKEd 2040 with an 11, that would have told the computer that the data for sprite zero was now to be found starting at memory location 832.

The colour of our sprite is determined by another memory location which we'll arrive at in a moment, after briefly mentioning one other variation on a sprite theme: multi-coloured sprites.

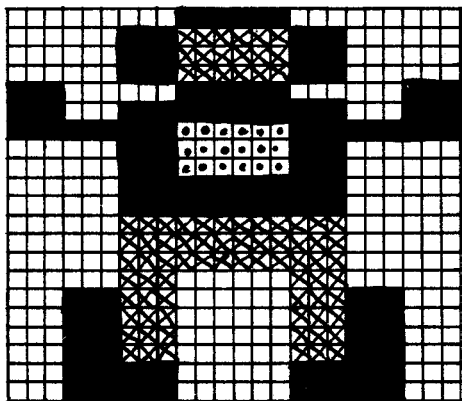
## **Multi-coloured sprites**

As you might surmise, a multi-coloured sprite is one that is capable of showing more than one colour, and indeed we can have up to four colours on display in such a sprite.

These are the ordinary sprite colour which we're already used to, the

background colour of the screen, and two other colours referred to as sprite multi-colour zero and sprite multi-colour one. We'll see shortly where this information is kept in the computer's memory.

With this feature available, why should anyone ever want to use ordinary sprites? Because a multi-coloured sprite may still occupy 63 bytes in memory, but each row of 24 by 21 bytes is now read in a somewhat different manner.



As you can see from the above, if we look at each row of data we have 24 bits to play with. Using an ordinary sprite, we can say that each of those bits will be either on or off. However, using a multi-coloured sprite we cannot look at each bit at once, since we need to know whether each multi-coloured sprite bit is any one of three colours, or is turned off.

A bit cannot be in four different states, and so the bits are joined together in twos, since two bits can indeed be looked at in one of four different ways, as shown below:

OFF OFF : Displays screen background colour  
OFF ON : Displays sprite multi-colour zero  
ON OFF : Displays ordinary sprite colour  
ON ON : Displays sprite multi-colour one

So by looking at two bits at a time, we can select which colour to display that part of the sprite in. Thus, although we still have 24 bits to play with, as far as sprite resolution goes they are to be regarded as just 12 different blocks: half the resolution of an ordinary sprite.

Before we start getting a little more complicated, here's where all the sprite data is stored in memory.

## Sprite data map

Starting at location 53248, we have 47 registers at our disposal, and these are grouped as follows:

Location	Function
(53248 + )	
00	X co-ordinate position of sprite 0
01	Y co-ordinate position of sprite 0
02-15	Ditto for sprites 1 to 7.
16	Most significant bit of X co-ordinate position of all sprites. Used when moving sprites at right hand side of screen, when X co-ordinate would normally be greater than 255
17	Used in selecting extended colour mode, scrolling screen in Y-direction, etc.
18	Raster register
19	X co-ordinate position of light pen
20	Y co-ordinate position of light pen
21	Turning selected sprites on
22	Used in selecting multi-coloured mode, scrolling screen in X-direction, etc.
23	Expand selected sprite in Y direction
24	Memory pointers
25	Interrupt registers

Location	Function
26	Enable interrupt
27	Sprite data priority: controls what happens when sprites hit something on the screen
28	Used in selecting multi-colour sprites
29	Expand selected sprite in X direction
30	Controls what happens when sprites collide with each other
31	Controls sprite to data collisions in conjunction with register 27
32	Controls the screen border colour
33	Controls screen background colour
34	Background colour 1, used in high resolution modes
35	Background colour 2, as above
36	Background colour 3, as above
37	Selects sprite multi-colour zero
38	Selects sprite multi-colour one
39	Selects colour for sprite 0
40-46	Ditto for sprites 1 to 7.

So, as you can see, with ordinary sprites you can have each of the eight sprites displayed in a different colour, whereas multi-coloured sprites have all got to have the same colour, since only one register is set aside for each multi-colour mode.

## Defining sprites in Basic

Later on in this book there are a number of programs which make life a lot easier when it comes to defining sprites.

But for the time being we're going to need to do it by hand, if only so that we can get a better grasp of what precisely is going on. Using someone else's programs is all very well, but you may end up not knowing anything about how the machine itself handles everything.

So for now we're going to do this the long way, by getting sheets of paper out and drawing all over them. I suggest that your first exercises are done using a pencil and rubber, since mistakes are bound to occur!

First of all, we'll need to draw up a 24 pixel by 21 pixel grid, like this:

```
ABCDEFGHIABCDEFGHIABCDEFGHI
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
```

Note the letters across the top. These are given the usual binary notation, so that A = 128, B = 64, and so on until we reach H, which is equal to 1. This will enable us to calculate the necessary data for each sprite, as we shall see.



Now, define a sprite! Let's say we came up with something like this:

[illegible]

A rather primitive looking rocket, but it will serve the purpose. To get the bytes of data for this sprite we must add up the values for each row of the sprite, so that the first row becomes equal to 0,24,0 (the three groups of letters).

We then carry on for each row, adding up all the numbers, until we get to the last row, in this case equal to 62,126,120.

Having now got all our 63 bytes of data, these have got to be POKEd into the relevant bit of memory, and the following program will do just that. for a rather different looking sprite: a Duckworth duck!

```

10 V=53248 : REM START OF VIDEO CHIP
20 FORI=0TO62:READA:POKE832+I,A:NEXT
1000 DATA 0,0,0,0,3,128,0
1010 DATA 4,64,0,9,112,0,8
1020 DATA 56,0,8,112,0,8,128
1030 DATA 78,9,0,177,249,0,100
1040 DATA 80,128,170,44,64,101,152
1050 DATA 32,19,0,32,8,0,64
1060 DATA 7,255,128,0,108,0,0
1070 DATA 108,0,0,111,0,0,110
1080 DATA 0,0,120,0,0,112,0

```

Incidentally these data statements were generated by the sprite designer program given later, albeit with different line numbers.

Now that we've worked out what all our data is, how do we actually begin to start using it?

## **Sprite data, and how it's stored**

The last program POKEd 63 bytes of data into memory locations 832 to 894, and this data will be used to draw up our sprite, but first we need to tell the computer that this is where we've stored all our data.

As we already know, memory locations 2040 to 2047, for sprites 0 to 8, tell the computer where to go, and assuming we're going to define this sprite to be sprite number 0, we need to alter memory location 2040. Sprite number 1 would look for location 2041, and so on.

Since we've put all our data in the 13th block of memory set aside for sprite data, there is little point in telling the computer that we've put it somewhere else, so we need to add:

```
13 POKE 2040,13
```

Now we have to turn the sprite on, and give it a colour. First of all, we'll give it a colour. A look at the map shows that the colour of sprite zero is defined by the content of memory location 39, so if we want a yellow sprite we must add the following:

```
14 POKE V+39,7
```

Now to turn it on, using memory location  $V + 21$ . Remembering how binary numbers operate, POKEing a 1 into this location will turn on sprite 0, a 2 will turn on sprite 1, a 4 for sprite 2 (or a 3 for sprites 0 and 1), and so on until we reach 255, which turns every sprite in the universe on (or at least all those defined in the computer's memory).

So the next line to add becomes:

```
15 POKE V+21,1
```

All we need to do now is to move the sprite about the screen, so that you can see your creation in action, and to do this we'll just update the X and Y co-ordinates ( $V + 0$  for X co-ord,  $V + 1$  for Y co-ord for sprite 0), like this:

```

30 FORI=0TO200
35 POKE V+0,I
40 POKE V+1,I
50 NEXT I
55 GOTO 30

```

This will send your Duckworth duck scurrying about the screen from the top left hand corner to somewhere near the bottom right, moving one pixel position in either direction at a time.

Of course we can get a bit more sophisticated than this, and the following program shows a number of different coloured ducks whizzing about the screen.

To explain the listing briefly, our Duckworth duck sprite data is in line 63000 onwards, and is read in in line 62035. The two lines before that determine whether or not it's a multi-coloured sprite by looking at the first four items of data. If it isn't, we just jump to line 62035 and read all the data. The first part of the program merely moves the four of them randomly about the screen.

```

5 POKE 53281,1:POKE 53280,12
10 T=4:GOSUB62000
15 PRINT"[CLR]"
20 FORI=0TO3
22 P=INT(RND(.5)*15):IFP=1THENP=8
23 POKE V+I+39,P
25 POKE V+I*2,INT(RND(.5)*220+30)
26 POKE V+I*2+1,INT(RND(.5)*150+50)
27 NEXTI
30 POKE V+21,255
40 FORK=1TO250:NEXTK:GOTO20
61999 END
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=0THENRETURN
62015 FORA=1TONS
62020 READSK,M1,M2:IFSK=0THENPOKEV+28,PEEK(V+28)AND
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READC0:POKEV+38+A,C0:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READQ:POKEC,Q:N
EXT:RESTORE:NEXT:RETURN
63000 DATA 0,2,7,8
63001 DATA 0,0,0,3,128,0
63002 DATA 4,64,0,9,112,0,8
63003 DATA 56,0,8,112,0,8,128
63004 DATA 78,9,0,177,249,0,100

```

```
63005 DATA0,128,170,44,64,101,152
63006 DATA32,19,0,32,8,0,64
63007 DATA7,255,128,0,108,0,0
63008 DATA108,0,0,111,0,0,110
63009 DATA0,0,120,0,0,112,0,0
```

## Multi-colour and expanded sprites

As always, it's swings and roundabouts time, as our sprite is now 24 by 21 pixels, but with the horizontal pixels joined up in pairs, thus giving us the ability to have four different colours assigned to each sprite.

The colours are defined in the usual bit pair sequence, with each pair taking on the following values:

- 00 : becomes transparent, and displays the screen colour.
- 01 : sprite multi-colour register 0 (53285)
- 10 : sprite ordinary colour register (53287-)
- 11 : sprite multi-colour register 1 (53286)

Sprites are expanded in the X direction with the following command:

```
POKE 53277,PEEK(53277)OR(2 to the power SN)
```

where SN is the sprite number from 0 to 7.

and in the Y direction with:

```
POKE 53271,PEEK(53271)OR(2 to the power SN)
```

To get life back to normal again, in the X direction:

```
POKE 53277,PEEK(53277)AND(255-(2 to the power SN))
```

and in the Y direction:

```
POKE 53271,PEEK(53271)AND(255-(2 to the power SN))
```

## Sprite positioning

So far we've never moved sprites beyond an X co-ordinate of 255, simply because memory locations can't hold values greater than this.

However, memory location 53264 allows us to move all the way to

the edge, in the following way.

When the X co-ordinate becomes equal to 255, POKE 53264 (or V + 16), with a 1, and then reset the X values to zero again. Now we're only moving from 256 to 320, or a total of 64 positions, so X ranges from 0 to 63. When moving back again, reset V + 16 back to a zero, let X equal 255 and control the sprite in the normal manner.

## Sprite priority and collision

We'll leave you to experiment with the program listings given elsewhere to see precisely how this works, but in brief the priority of each sprite can be controlled from register 53275 (53248 + 27).

This register works in exactly the same way as all the others, with sprite 0 being controlled from bit 0, sprite 1 from bit 1, and so on. If the bit is set to zero, then the sprite will be displayed on top of anything else: the sprite is in the foreground, in other words.

To get the relevant sprite into the background, the bit must be set to 1.

### Collision

This is controlled from memory location 53278, or 53248 + 30.

Again, this works in the same way as all the other locations, and is used to detect collisions between sprites.

If the register is showing zero, then nothing has happened; a 3 indicates a collision between sprites 0 and 1; a 6 for sprites 1 and 2, and so on.

This is based on the usual manner of selecting sprites from the appropriate bits of a particular byte.

i.e. Value	128	64	32	16	8	4	2	1
Bit	7	6	5	4	3	2	1	0
Sprite No.	7	6	5	4	3	2	1	0

Thus sprites 2 and 3 are controlled from bits 2 and 3, which respectively give the values of 4 and 8, and therefore a value of 12 (4 + 8) must be POKEd into that byte, or indeed read from it, and the relevant action will follow.

Multiple sprite collision is also possible from this.

For instance, if register 53278 returns a value of 82, it means that bits 6, 4 and 1 have been affected, or in other words sprites 6, 4 and 1 are involved in a collision.

A most useful location!

## Turning sprites off

Well, we'll have to get rid of them sometime! The quick and easy way to turn them all off is to type POKE V + 21, 0, but for selected sprites you must use:

POKE V + 21, PEEK(V + 21) AND (255 - 2 to the power of SN)

where SN is the sprite number from 0 to 7.

## Sprite movement

We've already shown you sprites moving across the screen, and you should now be in a position to amend the games listings given in this chapter to incorporate your own sprites.

Just define the sprites first, and then POKE the appropriate values into the X and Y co-ordinate locations as the nature of the game dictates.

Of course, this game could also be adapted for control by a joystick, using the following locations to test for movement, firing and so on.

For joystick in port 1:

S1 = PEEK(56321)

-(S1 AND 16) = 0 gives a 1 if the fire button is pressed, and a 0 if it's not.

((S1 AND 15) = 4) - ((S1 AND 15) = 8) gives a 1 for moving left, a -1 for moving right, and a 0 if nothing's doing.

((S1 AND 15) = 1) - ((S1 AND 15) = 2) gives a 1 for moving down, a -1 for moving up, and a 0 if nothing's doing.

To read a joystick in port 2, let S2 = 56320, and substitute S2 for

S1 in all of the above expressions.

Having done that, it would be a relatively simple matter to have a sprite controlled joystick game, written entirely in Basic.

None of the programs that follow have been designed with sprites in mind, although it wouldn't be too difficult to convert any of them to respond to a joystick input rather than a keyboard input.

## **Space Battle**

This first program, a fairly simple game, was designed to show how up to four sprites could be moved around the screen at the same time, while also providing a reasonably challenging game.

The object of the game is to stop the aliens (two sprites) from reaching the bottom of the screen by manoeuvring your spaceship around and firing at them. Your spaceship is, needless to say, another sprite, and a fourth sprite appears in the form of the missile which you launch against the aliens.

The whole game is played on a starry background, with the noise tone selected from the waveforms available indicating the descent of the aliens.

And now, a few notes about the program.

### **Program notes**

We won't go through every single line, but instead indicate routines of some importance, and also point out where anything interesting is happening.

Line 8 : clear the sound channel, and set the volume to its highest level.

Line 9 : set various sound parameters (see chapter eight).

Line 10 : set x position of sprite 0 (you), and go to subroutines to set up other sprites and give instructions.

Lines 20-40 : sprite parameters.

Lines 45-60 : which key pressed (SP denotes the speed of the game).

Lines 100-104 : moving left!

Lines 200-206 : moving right!

Lines 300-316 : move missile if fired.

Lines 400-412 : generate or update aliens.

Lines 414-421 : check progress of aliens, and change colour, and/or expand, if necessary.

Lines 500-520 : check for collision between alien and missile.

Lines 600-620 : oops!

Lines 2000-2200 : instructions for playing.

Lines 62000-62035 : get sprite data.

Lines 63000-63345 : the data itself.

```
8 S=54272:FORI=1TO24:POKE S+I,0:NEXT:POKE S+24,15
9 POKE S+5,5:POKE S+6,108:POKE S+12,5:POKE S+13,10
8:POKE S+4,129:POKE S+11,129
10 T=4:X=160:GOSUB62000:GOSUB2000
20 REM START OF GAME PROPER
21 POKE V+16,0
22 POKE V+21,0
25 POKE V+21,255
26 POKE V+4,0:POKE V+5,0
30 POKE V+1,227:POKE V,X
40 REM
45 IFPEEK(197)=10THEN100:REM MOVE LEFT
50 IFPEEK(197)=18THEN200:REM MOVE RIGHT
60 IFPEEK(197)=1ANDF=0THENF=1:SP=SP+.2:GOTO300:REM
  FIRE
65 IFF=1THEN310
70 GOSUB500:GOSUB510:GOTO 400:REM NOTHING DOING, S
  O UPDATE ALIENS
100 X=X-10-INT(SP):IFX<25ANDPEEK(V+16)=0THENX=25
101 IFX<1THENPOKEV+16,0:X=254
102 POKE V,X:IFF=1THEN310
103 GOTO 400
104 GOTO 40
200 X=X+10+INT(SP):IFX>254THENPOKE V+16,1:X=0
```



```

202 IFPEEK(V+16)=1ANDX>60THENX=60
204 POKE V,X:IFF=1THEN310
205 GOTO 400
206 GOTO 40
300 IFF=1THEN302
301 IFF=0THEN40
302 X1=X:Y1=207:IFPEEK(V+16)=1THENPOKE V+16,5
304 POKE V+4,X1:POKE V+5,Y1
310 Y1=Y1-20-INT(SP):IFY1<25THENY1=255:POKEV+5,Y1:
F=0:GOTO 40
312 GOSUB500:GOSUB510
314 POKE V+5,Y1
316 GOSUB510:GOSUB500
400 REM ALIENS
402 IFAP=1THEN407
403 AX=INT(RND(.5)*230+25):AY=20
404 POKE V+2,AX:POKE V+3,AY:AP=1:POKE V+40,6:POKE
V+29,0
407 IFAM=1THEN410
408 AA=INT(RND(.5)*230+25):AB=0
409 POKE V+7,AB:POKE V+6,AA:AM=1:POKE V+42,4:POKE
V+29,0
410 AY=AY+2+INT(SP):POKE V+3,AY:AB=AB+2+INT(SP):PO
KEV+7,AB
412 POKE S,AY:POKES+1,INT(AY/4):POKE S+7,AB:POKE S
+8,INT(AB/4)
414 IF AY>220THEN600:REM END OF GAME!
415 IFAY>120THENPOKEV+40,7
416 IFAY>160THENPOKE V+40,9:IFPEEK(V+29)=0THENPOKE
V+29,2
417 IFAY>160ANDPEEK(V+29)=8THENPOKE V+29,10
418 IF AB>220THEN600:REM END OF GAME!
419 IFAB>120THENPOKEV+42,3
420 IFAB>160THENPOKE V+42,1:IFPEEK(V+29)=0THENPOKE
V+29,8
421 IFAB>160ANDPEEK(V+29)=2THENPOKEV+29,10
450 GOTO 40
500 IFPEEK(V+30)=246THENAP=0:POKEV+3,0:POKEV+2,0:Y
1=10:AK=AK+1
502 IFPEEK(V+30)=254THENAP=0:POKEV+3,0:POKEV+2,0:Y
1=10:AK=AK+1
505 RETURN
510 IFPEEK(V+30)=252THENAM=0:POKEV+7,0:POKE V+6,0:
Y1=10:AK=AK+1
520 RETURN
600 POKE V+21,0:PRINT"[CLR]YOU BLEW IT!!"
602 PRINT"[2CD,CL]"AK" ALIENS KILLED !!"
604 POKE S+4,129:POKE S+11,129:POKE S,10:POKES+1,3
0:POKE S+7,10:POKE S+8,30
606 FORI=1TO10:POKE 53280,INT(RND(.5)*16):FORJ=1TO
100:NEXTJ,I:POKE 53280,9
608 PRINT"[WHT,2CD]PRESS 'SPACE' TO START AGAIN."

```

```

610 GET GU$:IFGU$<>" "THEN610
612 PRINT"[2CD]OK, JUST HANG ON!"
620 RUN
699 STOP
1999 END
2000 REM INSTRUCTIONS
2010 POKE 53280,9:POKE 53281,0
2012 PRINT"[CLR,WHT]WELCOME TO [YEL]SPACE INVASION
!!"
2014 PRINT"[2CD,WHT]JUST STOP THE INVADERS FROM RE
ACHING THEBOTTOM OF THE SCREEN."
2015 PRINT"[2CD]PRESS 'A' TO MOVE LEFT,'D' TO MOVE
RIGHTAND 'RETURN' TO FIRE."
2016 PRINT"[2CD]BUT BE WARNED : THE ACTION IS SLOW
LY, BUT SURELY, GETTING FASTER!"
2017 PRINT"[2CD]PRESS 'SPACE' TO START"
2018 GETKE$:IFKE$<>" "THEN2018
2020 PRINT"[CLR]"
2025 FORI=1TO75:PL=INT(RND(.5)*800)
2026 POKE 1024+PL,46:POKE 55296+PL,INT(RND(.5)*16)
2027 NEXT
2200 RETURN
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=0THENRETURN
62015 FORA=1TONS
62020 READSK,M1,M2:IFSK=0THENPOKEV+28,PEEK(V+28)AN
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READCD:POKEV+38+A,CD:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READQ:POKEC,Q:N
EXT:NEXT:RETURN
63000 DATA 0 , 2 , 7 , 1
63005 DATA0,24,0,0,60,0,0,102,0,0,219,0,1,189,128,
1
63010 DATA219,128,1,231,128,1,255,128,1,255,128,3,
255,192,7,255
63015 DATA224,15,219,240,31,255,248,63,255,252,63,
219,252,63,255,252
63020 DATA60,195,60,56,255,28,16,126,8,16,60,8,0,2
4,0,0
63025 DATA 0 , 2 , 7 , 6
63030 DATA0,0,0,0,0,0,0,0,0,195,0,1,231,128,7
63035 DATA255,224,15,255,240,31,0,248,63,255,252,1
07,255,214,232,0
63040 DATA23,107,255,214,63,255,252,31,0,248,15,25
5,240,7,255,224
63045 DATA0,195,0,0,195,0,0,195,0,0,195,0,0,195,0,
0
63200 DATA 0 , 0 , 0 , 12

```

```

63205 DATA0,48,0,0,48,0,0,48,0,0,120,0,0,252,0,1
63210 DATA254,0,1,254,0,1,254,0,1,254,0,3,3,0,6,1
63215 DATA128,6,1,128,6,1,128,6,1,128,0,0,0,0,0,0
63220 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
63325 DATA 0 , 2 , 7 , 4
63330 DATA0,0,0,0,0,0,0,0,0,0,195,0,1,231,128,7
63335 DATA255,224,15,255,240,31,0,248,63,255,252,1
07,255,214,232,0
63340 DATA23,107,255,214,63,255,252,31,0,248,15,25
5,240,7,255,224
63345 DATA0,195,0,0,195,0,0,195,0,0,195,0,0,195,0,
0

```

## Notes

There are no graphics other than sprites used in this game, so there's nothing to worry about there. The only thing to watch out for is the strange symbol in lines 62020 and 62025, which is meant to be an up-arrow (raising to the power of) symbol, but my printer just wouldn't have it!

## Sprite Generator

Now that you've seen a few sprites in action, and hopefully designed a few of your own, we'll make it a lot easier for you by giving you a listing for a sprite generator. This one is for single-coloured sprites only, but there will be a multi-coloured one a little later on.

There is some data already in there for displaying a sprite on the screen (it's our old friend the Duckworth duck again), so that you can see what's happening.

Using this program you can define sprites to your heart's content, move them about the screen, convert them to Basic data statements, expand or contract them, and do just about everything you'd legally want to do to a sprite.

## Program notes

This one is pretty heavily REMmed, so it shouldn't be too difficult to see what's happening. Nevertheless, we'll go through it all anyway.

Line 20 : using spare space in cassette buffer, indicate position of first sprite.

Lines 50-60 : read data, and if there is some go and draw up sprite.

Line 90 : function for keeping track of cursor on screen.

Lines 100-160 : sprite parameters.

Lines 180-290 : put up screen display.

Line 300 : and display program options.

Lines 330-570 : check for user update.

Lines 580-650 : point added, so update screen display and sprite display.

Lines 660-720 : editing another sprite!

Lines 730-800 : point removed, so again update both screen and sprite display.

Lines 810-850 : set up initial sprite, if any.

Lines 860-880 : define array as used in sprite calculation.

Lines 900-990 : expanding/contracting sprite in either X or Y direction.

Lines 1000-1140 : on-screen instructions.

Lines 1150-1200 : clear sprite and update screen display (by pressing shift and clear home keys).

Lines 1210-1390 : sprite goes walkabout, so check X and Y registers, and don't let it go too far off screen.

Lines 1400-1440 : sprite colour change.

Lines 1450-1590 : turn sprite data into actual data statements.

Lines 30000-30010 : the sprite data (a duck).

```

10 REM SPRITE GENERATOR
12 REM FROM AN ORIGINAL PROGRAM BY RICHARD FRANKLI
N
14 REM WELL DONE!
20 POKE 829,223
29 REM
30 REM IF ANY SPRITE DATA,SET UP SPRITE
31 REM IT LOOKS AN AWFUL LOT BETTER UNEXPANDED
40 POKE 828,0
50 READ SP
60 IF SP>0THEN 810
69 REM
70 REM NO MORE SPRITE DATA
71 REM
80 GOSUB 860:POKE53281,0:POKE53280,8:PRINT"[YEL]"
90 DEFFNA(ZZ)=1064+R*40+C
100 V=53248:NO=PEEK(829)
110 XL=0:YL=1:XG=16:SE=21:XY=23:XX=29
120 SC=39:PRINT"[CLR]"
130 POKE 2040,NO:POKE V+SE,1:POKE V+XY,1
140 POKE V+XX,1:POKE V+XL,255:POKE V+YL,190
150 POKE V+XG,0
160 X=255:Y=190
169 REM
170 REM SET UP DISPLAY
171 REM
180 PRINT"[HOME,GRN]X      X      X      X":REM
USES SHIFTED X FOR GRAPHICS
185 LOC=64*NO:PRINT"[HOME]"
190 FORI=LOCTOLOC+62STEP3
200 FORJ=0TO2
210 ZZ=PEEK(I+J)
220 FORK=7TO0STEP-1
230 A=INT((ZZANDAX(K))/AX(K))
240 IFA=1THENPRINT"[YEL]Q";:GOTO260:REM USES SHIFT
ED Q FOR GRAPHICS
250 PRINT"[YEL].";
260 NEXTK
270 NEXTJ
280 PRINT
290 NEXTI
300 GOSUB1000
309 REM
310 REM SPRITE SET UP ON THE SCREEN
320 REM INPUT CHANGES
321 REM
330 R=0:C=0
340 Z=FNA(0)
342 PC=PEEK(Z):PK=Z
345 POKE Z,24:
350 POKEZ+54272,1
360 GETA$: IFA$=""THEN360

```

```

370 POKEZ+54272,7:POKEPK,PC
380 IFA$="Q"THENPRINT"[CLR]":END
390 IFA$="[CR]"ANDC=23THEN C=0:GOTO340
400 IFA$="[CR]"THENC=C+1:GOTO340
410 IFA$="[CL]"ANDC=0THENC=23:GOTO340
420 IFA$="[CL]"THENC=C-1:GOTO340
430 IFA$="[CD]"ANDR=20THENR=0:GOTO340
440 IFA$="[CD]"THENR=R+1:GOTO340
450 IFA$="[CU]"ANDR=0THENR=20:GOTO340
460 IFA$="[CU]"THENR=R-1:GOTO340
470 IFA$="[HOME]"THENR=0:C=0:GOTO340
480 IFA$="[CLR]"THENGOSUB1150:GOTO340
490 IFA$="+"THEN580
500 IFA$="-"THEN730
510 IFA$="M"THEN1210
520 IFA$="B"THEN1450
530 IFA$="C" THEN 1400
540 IFA$="X"THEN900
550 IFA$="N"ANDNO-223<31THENNO=NO+1:GOTO130
560 IFA$="E"THEN660
570 GOTO 340
574 REM
575 REM ADD POINT
576 REM
580 Z=FNA(0)
590 Z1=PEEK(Z)
600 IFZ1=81THEN340
610 POKEZ,81
620 BYTE=INT(C/8)+R*3
630 BIT=7-(C-INT(C/8)*8)
640 POKEBYTE+NO*64,PEEK(BYTE+NO*64)ORA%(BIT)
650 GOTO 340
654 REM
655 REM INPUT SPRITE # TO EDIT
656 REM
660 INPUT"[HOME,23CD,7CR,RVS]SPRITE NO. [4C
L]";S
670 IFS<0ORS>31THEN660
680 IF NO=223+STHENZZ=1:GOTO700
690 NO=223+S
700 PRINT"[HOME,23CD,39SP,HOME]";
710 IFZZ=1THENZZ=0:GOTO340
720 GOTO 130
724 REM
725 REM DELETE POINT
726 REM
730 Z=FNA(0)
740 Z1=PEEK(Z)
750 IFZ1=46THEN 340
760 POKE Z,46
770 BYTE=INT(C/8)+R*3
780 BIT=7-(C-INT(C/8)*8)

```

```

790 POKE BYTE+NO*64,PEEK (BYTE+NO*64) AND (255-A%(BIT
))
800 GOTO 340
804 REM
805 REM IF ANY DATA, SET SPRITES UP
806 REM
810 LOC=SP*64
820 FOR I=LOC TO LOC+62
830 READ A:POKE I,A
840 NEXT I
842 READ CS
844 NO=PEEK (829)
845 POKE 53248+39+NO-223,CS
850 GOTO 50
854 REM
855 REM SET ARRAY WITH POWERS OF TWO
856 REM
860 FOR I=0 TO 7
870 A%(I)=2^I
880 NEXT I
890 RETURN
894 REM
895 REM INPUT FOR EXPAND
896 REM
900 PRINT"[HOME,23CD,10CR,RVS]ENTER X OR Y"
910 GETA$:IFA$<>"X"AND A$<>"Y"THEN 900
920 IFA$="X"THEN 960
930 IFPEEK (V+XY)=1THEN POKE V+XY,0:GOTO 980
940 POKE V+XY,1
950 GOTO 980
960 IFPEEK (V+XX)=1THEN POKE V+XX,0:GOTO 980
970 POKE V+XX,1
980 PRINT"[HOME,23CD,10CR,12SP]"
990 GOTO 340
994 REM
995 REM DISPAY CONTROL OPTIONS
996 REM
1000 PRINT"[HOME]"SPC(26)"[RVS]CONTROLS[OFF]"
1005 PRINTSPC(25)"SPRITE #[RVS,GRN]"NO-223
1010 PRINT:PRINTSPC(25)"[WHT,RVS]E[OFF]DIT SPRITE
#"
1020 PRINTSPC(25)"[RVS]N[OFF]EXT SPRITE #"
1030 PRINTSPC(25)"[RVS]M[OFF]OVE SPRITE"
1040 PRINTSPC(25)"[RVS]C[OFF]OLOUR CHANGE"
1050 PRINTSPC(25)"[RVS]X[OFF]PAND"
1060 PRINTSPC(25)"[RVS]+[OFF] ADD DOT"
1070 PRINTSPC(25)"[RVS]-[OFF] REMOVE DOT"
1080 PRINTSPC(25)"[RVS]B[OFF]ASIC DATA"
1090 PRINTSPC(25)"[RVS]Q[OFF]UIT"
1100 PRINT:PRINTSPC(25)"USE CURSOR"
1110 PRINTSPC(25)"CONTROL TO"
1120 PRINTSPC(25)"POSITION"

```

```

1130 PRINTSPC(25)"CURSOR."
1140 RETURN
1144 REM
1145 REM CLEAR PRESENT SPRITE
1146 REM
1150 FORI=0TO62:POKENO*64+I,0:NEXTI
1160 FORI=0TO20
1170 FORJ=0TO23
1180 POKE1064+I*40+J,46
1190 NEXTJ,I:R=0:C=0
1200 RETURN
1204 REM
1205 REM MOVE SPRITE AROUND SCREEN
1206 REM
1210 PRINT"[HOME,22CD,RVS]USE CURSOR KEYS TO MOVE
THE SPRITE,"
1220 PRINT"[RVS]RETURN TO RETURN TO EDITING"
1230 GETA$:IFA$=""THEN1230
1240 IFA$="[CR]"ANDX<319THENX=X+2
1250 IFA$="[CL]"ANDX>1THENX=X-2
1260 IFA$="[CD]"ANDY<254THENY=Y+2
1270 IFA$="[CU]"ANDY>1THENY=Y-2
1280 POKE V+YL,Y
1290 POKE V+XG,INT(X/255)
1300 POKE V+XL,X-INT(X/255)*255
1310 IF A$=CHR$(13)THEN1330
1320 GOTO1210
1330 POKE V+XL,255
1340 POKE V+YL,190
1350 POKE V+XG,0
1360 X=255:Y=190
1370 PRINT"[HOME,22CD,36SP]"
1380 PRINT"[36SP,HOME]"
1390 GOTO 340
1394 REM
1395 REM CHANGE SPRITE COLOUR
1396 REM
1400 INPUT"[HOME,23CD,9CR,RVS]COLOUR (0-15)
[5CL]";CO
1410 IF CO<0ORCO>15THEN1400
1420 POKE V+SC,CO
1430 PRINT"[HOME,23CD,39SP,HOME]";
1440 GOTO 340
1444 REM
1445 REM CREATE DATA STATEMENTS FOR
1446 REM PRESENT SPRITE
1447 REM
1450 PRINT"[CLR,3CD]";PEEK(828)+30000;"DATA"RIGHT$(
STR$(NO),LEN(STR$(NO))-1)
1460 POKE828,PEEK(828)+1:FORI=0TO8
1470 PRINTPEEK(828)+30000"DATA";
1480 FORJ=0TO6

```



```

1490 BB=PEEK(NO*64+I*7+J)
1500 BB$=RIGHT$(STR$(BB),LEN(STR$(BB))-1)
1510 PRINTBB$;" ";
1520 NEXT J
1530 PRINT"[CL] ";POKE828,PEEK(828)+1
1540 NEXT I
1550 PRINTPEEK(828)+30000;"DATA";C0;"",-1"
1560 PRINT"RUN80[HOME]"
1570 POKE 198,12
1580 FORI=0TO11:POKE631+I,13:NEXT I
1590 POKE829,NO:END
29997 REM
29998 REM SPRITE DATA STORED FROM HERE
29999 REM
30000 DATA223
30001 DATA0,0,0,0,3,128,0
30002 DATA4,64,0,9,112,0,8
30003 DATA56,0,8,112,0,8,128
30004 DATA78,9,0,177,249,0,100
30005 DATA80,128,170,44,64,101,152
30006 DATA32,19,0,32,8,0,64
30007 DATA7,255,128,0,108,0,0
30008 DATA108,0,0,111,0,0,110
30009 DATA0,0,120,0,0,112,0
30010 DATA 15 , -1

```

## Notes

There are only two graphic characters used in this program: the shifted X character in line 180, and the shifted Q in line 240.

Our problem with the up-arrow key surfaces again in line 870 (see notes for Space Battle above).

Apart from that, it shouldn't be too difficult to get this program up and running.

Continuing with my policy of letting you play about with a program before showing you the utility used to generate it, the next (very!) simple game was written to show how four different multi-coloured sprites could be used in a game to produce a kind of animation effect.

This is just one way of doing it, obviously there are others. However, the simple expedient of replacing one sprite with another slightly different one in exactly the same position does manage to produce the effect that we're after.

# Alpine Slopes

This really is a very straightforward program, but it does show what we were after: four different multi-coloured sprites in action according to the dictates of the person playing the game.

It's an old idea - you have to guide your little man down a ski-slope which zigzags about from side to side. Needless to say, if you hit the sides that is the end of the slope as far as you're concerned, and the game gives you a report of how long you managed to stay upright.

Just to make it that little bit harder, the course also gets narrower the longer the game progresses.

The four sprites used are all variations on the theme of a man on (or off, in one case) a pair of skis. In order of data, they show a little man skiing to the left, one skiing to the right, one skiing straight ahead (with a 'look, no hands' approach), and the fourth sprite is of a not very happy skier who's just collided with the side of the course.

## Program notes

Nothing of any great difficulty with this one, so here we go.

Line 10 : declare number of sprites, and then set sprites up and display instructions.

Line 12 : place X and Y co-ordinates at top centre of screen.

Line 16 : set up various sound parameters.

Line 20 : set up a primitive wall!

Line 25 : set up various sprite parameters.

Lines 30-32 : set up start of course.

Line 33 : set time at start of game.

Line 34 : set man up skiing straight ahead.

Lines 40-80 : which direction will he ski in? '1' indicates left, '2' indicates straight ahead, and '3' off to the right.

Lines 100-199 : moving left!

Lines 200-299 : and straight ahead.

Lines 300-399 : and to the right.

Line 500 : if the sprite's hit anything, that's the end of the game.

Line 502 : update ski-course.

Line 504 : back to where you came from.

Lines 500-599 : end of game, so start another one.

Lines 600-610 : check to see whether course needs narrowing, and which direction to move it in.

Lines 2000-2100 : instructions.

Lines 62000-62035 : read and set up sprite data.

Lines 63000-63520 : sprite data.

```
10 T=4:GOSUB62000:GOSUB2000
12 X=170:Y=60
16 POKE S+24,15:POKE S+5,40:POKE S+6,146:POKE S+4,
129:POKE S+3,12
18 PRINT"[GRN]
20 B$="XX XX":REM USE SHIFTED X FOR GR
APHICS
25 POKE V+21,4:POKE V+4,X:POKE V+5,Y:POKEV+31,0:P=
12
30 PRINT"[CLR]"TAB(P);B$
32 FORI=1TO20:PRINTTAB(P);B$:NEXTI
33 BB=TI
34 GOTO 200
40 GETA$
50 IFA$="1"THEN100
60 IFA$="2"THEN200
70 IFA$="3"THEN300
75 IFA$(">")THEN200
80 GOTO 40
100 POKE V,X:POKE V+1,Y
102 POKE V+21,1
```

```

103 X=X-8:Y=Y+1
104 GOSUB500
105 IFX<25THENX=25
106 POKE S+1,5:POKE S,20
107 IFY>180THENY=180
120 IFPEEK(197)=56THEN100
130 IFPEEK(197)=64THEN100
199 GOTO 40
200 POKE V+4,X:POKE V+5,Y
202 POKE V+21,4
203 X=X:Y=Y+1
204 GOSUB500
205 IFX<25THENX=25
206 POKE S+1,10:POKE S,30
207 IFY>180THENY=180
220 IFPEEK(197)=59THEN200
230 IFPEEK(197)=64THEN200
299 GOTO 40
300 POKE V+2,X:POKE V+3,Y
302 POKE V+21,2
303 X=X+8:Y=Y+1
304 GOSUB500
305 IFX>255THENX=255
306 POKE S+1,5:POKE S,40
307 IFY>180THENY=180
320 IFPEEK(197)=8THEN300
330 IFPEEK(197)=64THEN300
399 GOTO 40
500 IFPEEK(V+31)<>0THENPRINT"[HOME,GRN]KABOODMMMM!
!!!":GOTO550
502 GOSUB600
504 RETURN
550 POKE V+6,X:POKE V+7,Y:POKE V+21,8
551 POKE S+4,129:POKE S+1,20:POKE S,40
552 FORI=1TO2000:NEXT:POKE V+21,0
554 PRINT"[CLR]OH DEAR !!"
555 PRINT"[2CD]YOU LASTED ";INT((TI-BB)/60);"SECON
DS!"
560 POKE S+4,129:POKE S+1,20:POKE S,40
599 POKE V+31,0:RUN
600 A=A+1:IFLEN(B$)<10THEN602
601 IFA/30=INT(A/30)THENB$=LEFT$(B$,2)+MID$(B$,3,L
EN(B$)-5)+RIGHT$(B$,2)
602 IF(RND(.5)*10)>4.6THENP=P+1:GOTO606
604 P=P-1:IFP=-1THENP=0:GOTO608
606 IFP>19THENP=18
608 PRINTTAB(P)B$
610 RETURN
1999 END
2000 REM INSTRUCTIONS
2001 POKE V+21,4:POKE V+4,170:POKE V+5,52

```

```

2002 PRINT"[CLR]"
2003 S=54272:FORI=0TO24:POKE S+I,0:NEXT
2004 POKE 53281,1:POKE 53280,12
2006 PRINT"[2CD,8RN]WELCOME TO THE ANCIENT ART OF
SKI-ING."
2008 PRINT"[2CD]STEER YOUR LITTLE MAN DOWN THE TRA
CK BY USING THE 1,2 AND 3 KEYS ";
2010 PRINT"FOR MOVING LEFTSTRAIGHT, AND RIGHT RESP
ECTIVELY."
2012 PRINT"[2CD]AVOID THE SIDES OF THE TRACK FOR A
S LONGAS YOU CAN."
2014 PRINT"[2CD]PRESS 'SPACE' WHEN YOU'RE READY TO
START"
2016 GETKY$:IFKY$<>" THEN2016
2020 PRINT"[CLR]"
2100 RETURN
62000 V=53248
62005 B(0)=248:B(1)=249:B(2)=250:B(3)=251:B(4)=252
:B(5)=253:B(6)=254:B(7)=255
62010 NS=T:IFNS=0THENRETURN
62015 FORA=1TONS
62020 READSK,M1,M2:IFSK=0THENPOKEV+28,PEEK(V+28)AN
D255-2^(A-1):GOTO62030
62025 POKEV+28,PEEK(V+28)OR2^(A-1):POKEV+37,M1:POK
EV+38,M2
62030 READCD:POKEV+38+A,CD:POKE2039+A,B(A-1)
62035 FORC=B(A-1)*64TOB(A-1)*64+63:READQ:POKEC,Q:N
EXT:NEXT:RETURN
63000 DATA 1 , 2 , 7 , 8
63005 DATA0,80,0,0,144,0,0,144,0,0,80,0,1,84,0,1
63010 DATA244,0,5,212,0,21,116,0,81,84,0,65,84,0,2
,168
63015 DATA0,2,168,0,10,40,0,10,40,0,40,160,0,162,1
28,0
63020 DATA162,145,64,41,165,0,5,164,0,20,80,0,17,6
4,0,0
63025 DATA 1 , 2 , 7 , 8
63030 DATA0,5,64,0,6,64,0,6,64,0,6,64,0,21,64,0
63035 DATA31,64,0,23,80,0,21,84,0,31,69,0,21,64,0,
42
63040 DATA128,0,42,128,0,42,128,0,40,160,0,10,40,0
,2,138
63045 DATA1,74,40,0,105,160,0,20,80,0,5,20,0,1,69,
0
63050 DATA 1 , 2 , 7 , 8
63055 DATA0,20,0,0,105,0,0,105,0,4,20,16,4,85,16,5
63060 DATA125,80,0,125,0,0,85,0,0,85,0,0,170,0,0,1
70
63065 DATA0,0,170,0,0,130,0,1,130,64,1,130,64,1,13
0,64
63070 DATA1,130,64,1,130,64,1,65,64,1,65,64,1,65,6
4,0

```

```

63500 DATA 1 , 2 , 7 , 8
63505 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
63510 DATA 0,16,0,64,64,0,17,0,0,4,0,0,17,0,0,64
63515 DATA 64,1,0,16,0,0,0,0,0,0,160,0,10,40,0,40
63520 DATA 10,0,160,2,150,128,0,125,0,0,125,0,0,85,
0,0

```

## Notes

Nothing to note with this one really, as only one graphics character has been used. That is in line 20 where the italicised Xs are meant to show the shifted X character, the three leaf clover, otherwise known as the club symbol.

As before, we'll now give you the listing for the multi-colour sprite generator. The data that comes in it is meant to represent a sprite of someone pouring a pint of beer, but I think the end result failed to do justice to the idea. Still, the program's meant for your sprites, not mine, so we'll leave it to your imagination.

## Multi-Colour Sprite Designer

This follows much the same format as the original sprite designer program, with a number of important modifications to allow for the fact that we are using multi-coloured sprites now. The listing is given again in full, although you will find that quite a few lines are in common with that original listing.

However, on the theory that if we just gave the new lines and told you which lines to remove from the old listing then everyone would get totally confused, we present it here in full.

The lines to watch out for are 625 to 627, which are the ones used to calculate the update for the sprite itself, once the screen display has been updated by the user. These look and see what the two spaces on the screen represent, since we designated the three colours (forgetting background for a while), as a combination of spaces and shifted spaces. They may look the same on the screen to you, but to the program trying to keep the rest of the sprite the same as it was while updating the part you've chosen to alter, they look healthily different.

## **Program notes**

We'll go through the whole program again, to save you flipping back to the notes for the original sprite designer program and having half the pages in the book fall out.

Line 45 : set up parameters for video chip, and where sprite data will be stored.

Lines 50-60 : any data to be read?

Line 90 : function to keep track of cursor on screen (double cursor this time).

Lines 110-165 : set up various sprite parameters.

Lines 180-183 : set up screen display.

Line 184 : print on-screen instructions, and no I can't remember why I put in GOTO309 either now!

Lines 330-370 : update double cursor on screen.

Lines 380-570 : check on user input.

Lines 580-620 : point added, so update screen and sprite displays.

Lines 625-627 : checks state of old sprite before updating it.

Lines 730-770 : point removed, so update screen and sprite displays again.

Lines 808-850 : read any sprite data.

Lines 900-990 : expand or contract sprite in either X or Y direction.

Lines 1000-1140 : print on-screen options.

Lines 1150-1200 : remove sprite, so clear screen and sprite displays.

Lines 1210-1390 : sprite goes on sponsored walk.

Lines 1400-1440 : change drawing colour.

Lines 1450-1590 : convert sprite to Basic data statements.

Lines 3000-4004 : welcome to the show.

Lines 30011-30021 : sprite data for beer pouring.

```
10 REM MULTI-COLOUR SPRITE GENERATOR
15 REM ON FIRST RUN THROUGH, TRY SELECTING (IN ORD
ER), COLOURS 1,9 AND 12
20 REM FOR A GLIMPSE OF ONE OF THE AUTHOR'S FAVOUR
ITE SIGHTS!!
29 REM
30 REM IF ANY SPRITE DATA,SET UP SPRITE
31 REM IT LOOKS AN AWFUL LOT BETTER UNEXPANDED
40 POKE 828,0
45 V=53248:NO=13:SP=13:POKE V+21,0
50 READ FF
60 IF FF>0 THEN B04
69 REM
70 REM NO MORE SPRITE DATA
71 REM
80 POKE53281,0:POKE53280,8:PRINT"[YEL]"
85 GOSUB 3000
90 DEFFNA(ZZ)=1065+R*40+C
110 XL=0:YL=1:XG=16:SE=21:XY=23:XX=29
120 SC=39:PRINT"[CLR]"
130 POKE 2040,13:POKE V+SE,1:POKE V+XY,0
140 POKE V+XX,0:POKE V+XL,255:POKE V+YL,190
150 POKE V+XG,0
160 X=255:Y=190
165 POKE V+28,1
169 REM
170 REM SET UP DISPLAY
171 REM
180 PRINT"[HOME,YEL,SP,24CBMP]":REM PRESS CBM KEY
AND P FOR GRAPHIC CHARACTER
181 FORI=0TO20
182 PRINT"[CBMN,24SP,CBMH]":REM PRESS CBM N AND CB
M H FOR GRAPHICS CHARACTERS
183 NEXTI:PRINT"[SP,24CBMY]":REM PRESS CBM KEY AND
Y FOR GRAPHICS CHARACTER
184 GOSUB1000:GOTO309
309 REM
310 REM SPRITE SET UP ON THE SCREEN
320 REM INPUT CHANGES
321 REM
330 R=0:C=0
340 Z=FNA(0)
342 PC=PEEK(Z):PD=PEEK(Z+1):PK=Z
343 IFPC=160 THENRV=128:GOTO345
```



```

344 RV=0
345 POKE Z,24+RV:POKE Z+1,24+RV
346 KN=PEEK(Z+54272):KM=PEEK(Z+54273):IFKN=0THENKN
=1:KM=1
350 POKEZ+54272,KN:POKE Z+54273,KM
360 GETA$:IFA$=""THEN360
370 POKEZ+54272,KN:POKE Z+54273,KM:POKEPK,PC:POKEP
K+1,PD
380 IFA$="Q"THENPRINT"[CLR,CD]I HOPE YOU'VE TURNED
YOUR SPRITE INTO DATA!":END
390 IFA$="[CR]"ANDC=22THEN C=0:GOTO340
400 IFA$="[CR]"THENC=C+2:GOTO340
410 IFA$="[CL]"ANDC=0THENC=22:GOTO340
420 IFA$="[CL]"THENC=C-2:GOTO340
430 IFA$="[CD]"ANDR=20THENR=0:GOTO340
440 IFA$="[CD]"THENR=R+1:GOTO340
450 IFA$="[CU]"ANDR=0THENR=20:GOTO340
460 IFA$="[CU]"THENR=R-1:GOTO340
470 IFA$="[HOME]"THENR=0:C=0:GOTO340
480 IFA$="[CLR]"THENGOSUB1150:GOTO340
490 IFA$="+ "THENS80
500 IFA$="- "THEN730
510 IFA$="M"THEN1210
520 IFA$="B"THEN1450
530 IFA$="C" THEN 1400
540 IFA$="E"THEN900
570 GOTO 340
574 REM
575 REM ADD POINT
576 REM
580 Z=FNA(0)
590 IFCS=A2THENK1=224:K2=160
595 IFCS=A4THENK1=160:K2=224
600 IFCS=A3THENK1=160:K2=160
610 POKEZ,K1:POKE Z+1,K2:POKE55296+Z-1024,CS:POKE
55296+Z-1023,CS
620 BYTE=INT(C/8)+R*3:GOSUB625:GOTO340
625 CH=1065+40*R+8*INT(C/8):TC=0:BI=8
626 FORL1=CHTOCH+7:P=PEEK(L1):BI=BI-1:IFP=160ORP=2
4THENTC=TC+2^BI
627 NEXTL1:POKEBYTE+NO*64,TC:RETURN
724 REM
725 REM DELETE POINT
726 REM
730 Z=FNA(0)
740 Z1=PEEK(Z)
760 POKE Z,32:POKE Z+1,32
770 BYTE=INT(C/8)+R*3:GOSUB625:GOTO340
804 REM
805 REM IF ANY DATA, SET SPRITES UP
806 REM
808 READ A1,A2,A3:IFA1=0THENPOKE V+28,PEEK(V+28)AN

```

```

D255-2^(0):GOTO810
809 POKE V+28,PEEK(V+28)OR2^(0):POKE V+37,A2:POKE
V+38,A3
810 READ A4:POKE V+39,A4
815 LOC=SP*64
820 FOR I=LOC TO LOC+62
830 READ A:POKE I,A
840 NEXT I
850 GOTO50
894 REM
895 REM INPUT FOR EXPAND
896 REM
900 PRINT"[HOME,23CD,10CR]PRESS X OR Y"
910 GETA$: IFA$<>"X"ANDA$<>"Y"THEN900
920 IFA$="X"THEN960
930 IFPEEK(V+XY)=1THENPOKEV+XY,0:GOTO980
940 POKEV+XY,1
950 GOTO980
960 IFPEEK(V+XX)=1THENPOKEV+XX,0:GOTO980
970 POKEV+XX,1
980 PRINT"[HOME,23CD,10CR,12SP]
990 GOTO 340
994 REM
995 REM DISPAY CONTROL OPTIONS
996 REM
1000 PRINT"[HOME,YEL]"SPC(27)"[RVS]CONTROLS[OFF]"
1005 PRINTSPC(26)"SPRITE # 0"
1030 PRINTSPC(26)"[RVS]M[OFF]OVE SPRITE"
1040 PRINTSPC(26)"[RVS]C[OFF]HANGE COLOUR"
1050 PRINTSPC(26)"[RVS]E[OFF]XPAND"
1060 PRINTSPC(26)"[RVS]+[OFF] ADD DOT"
1070 PRINTSPC(26)"[RVS]-[OFF] REMOVE DOT"
1080 PRINTSPC(26)"[RVS]B[OFF]ASIC DATA"
1090 PRINTSPC(26)"[RVS]Q[OFF]UIT"
1100 PRINT:PRINTSPC(26)"USE CURSOR"
1110 PRINTSPC(26)"CONTROL TO"
1120 PRINTSPC(26)"POSITION"
1130 PRINTSPC(26)"CURSOR."
1140 RETURN
1144 REM
1145 REM CLEAR PRESENT SPRITE
1146 REM
1150 FORI=0TO62:POKEND*64+I,0:NEXTI
1160 FORI=0TO20
1170 FORJ=1TO24
1180 POKE1064+I*40+J,32
1190 NEXTJ,I:R=0:C=0
1200 RETURN
1204 REM
1205 REM MOVE SPRITE AROUND SCREEN
1206 REM
1210 PRINT"[HOME,23CD]USE CURSOR KEYS/RETURN TO EX

```

```

IT."
1230 GETA$: IFA$="" THEN 1230
1240 IFA$="[CLR]" AND X<319 THEN X=X+2
1250 IFA$="[CL]" AND X>1 THEN X=X-2
1260 IFA$="[CD]" AND Y<254 THEN Y=Y+2
1270 IFA$="[CU]" AND Y>1 THEN Y=Y-2
1280 POKE V+YL,Y
1290 POKE V+XG,INT(X/255)
1300 POKE V+XL,X-INT(X/255)*255
1310 IF A$=CHR$(13) THEN 1330
1320 GOTO 1210
1330 POKE V+XL,255
1340 POKE V+YL,190
1350 POKE V+XG,0
1360 X=255:Y=190
1370 PRINT"[HOME,23CD,39BP]"
1390 GOTO 340
1394 REM
1395 REM CHANGE SPRITE COLOUR
1396 REM
1400 PRINT"[HOME,23CD,YEL]PRESS 1, 2 OR 3 TO MAKE
YOUR CHOICE."
1402 GETCB$: IFCB$="" THEN 1402
1403 IFCB$="1" THEN CS=A2:GOTO 1420
1404 IFCB$="2" THEN CS=A3:GOTO 1420
1405 IFCB$="3" THEN CS=A4:GOTO 1420
1406 GOTO 1402
1420 POKE V+36+VAL(CB$),CS
1430 PRINT"[HOME,23CD,38SP,HOME";
1440 GOTO 340
1444 REM
1445 REM CREATE DATA STATEMENTS FOR
1446 REM PRESENT SPRITE
1447 REM
1450 PP=PP+11:PRINT"[CLR,3CD]";PP+30000;"DATA";SP;
",1,";A2;",";A3;",";A4;"
1460 FOR I=0 TO 8
1470 PRINT PP+I+30001;"DATA";
1480 FOR J=0 TO 6
1490 BB=PEEK(NO*64+I*7+J)
1500 BB$=RIGHT$(STR$(BB),LEN(STR$(BB))-1)
1510 PRINT BB$;",";
1520 NEXT J
1530 PRINT"[CL]"
1540 NEXT I
1550 PRINT PP+10+30000;"DATA -1";NO=NO+1:SP=SP+1
1560 PRINT"GOTO 45[HOME]"
1570 POKE 198,12
1580 FOR I=0 TO 11:POKE 631+I,13:NEXT I
1590 END
3000 PRINT"[CLR]MULTI-COLOUR SPRITE DESIGNER."
3002 PRINT"[2CD]YOU CAN DISPLAY UP TO FOUR COLOURS

```

```

PER SPRITE NOW. THESE ARE :="
3004 PRINT"[2CD]SPRITE MULTI-COLOUR ONE,
3006 PRINT"SPRITE MULTI-COLOUR TWO,
3008 PRINT"ORDINARY SPRITE COLOUR,"
3010 PRINT"AND THE SCREEN COLOUR."
3012 PRINT"[2CD]THIS LIMITS OUR SPRITE RESOLUTION
TO 12 BY 21 CHARACTERS.*
3014 PRINT"[2CD]NEVERTHELESS, WITH A LITTLE BIT OF
IMAGINATION, SOME ";
3016 PRINT"STAGGERING RESULTS CANBE ACHIEVED."
3018 GOSUB 4000
3019 POKE 53281,1
3020 PRINT"[CLR,YEL]FIRST OF ALL, YOU MUST CHOOSE
YOUR COLOURS FROM THE ";
3022 PRINT"16 AVAILABLE."
3023 PRINTTAB(30)CHR$(18)"[BLK] [OFF,GRN] - 0"
3024 PRINTTAB(30)CHR$(18)"[WHT] [OFF,GRN] - 1"
3025 PRINTTAB(30)CHR$(18)"[RED] [OFF,GRN] - 2"
3026 PRINTTAB(30)CHR$(18)"[CYN] [OFF,GRN] - 3"
3027 PRINTTAB(30)CHR$(18)"[PUR] [OFF,GRN] - 4"
3028 PRINTTAB(30)CHR$(18)"[GRN] [OFF,GRN] - 5"
3029 PRINTTAB(30)CHR$(18)"[BLU] [OFF,GRN] - 6"
3030 PRINTTAB(30)CHR$(18)"[YEL] [OFF,GRN] - 7"
3031 PRINTTAB(30)CHR$(18)"[ORG] [OFF,GRN] - 8"
3032 PRINTTAB(30)CHR$(18)"[BRN] [OFF,GRN] - 9"
3033 PRINTTAB(30)CHR$(18)"[LT.RED] [OFF,GRN] -
10"
3034 PRINTTAB(30)CHR$(18)"[GREY1] [OFF,GRN] -
11"
3035 PRINTTAB(30)CHR$(18)"[GREY2] [OFF,GRN] -
12"
3036 PRINTTAB(30)CHR$(18)"[LT.GRN] [OFF,GRN] -
13"
3037 PRINTTAB(30)CHR$(18)"[LT.BLU] [OFF,GRN] -
14"
3038 PRINTTAB(30)CHR$(18)"[GREY3] [OFF,GRN] -
15"
3040 INPUT "SPRITE MULTI-COLOUR ZERO";A2:IFA2<0ORA
2>15THENPRINT"[2CU]":GOTO3040
3042 INPUT "SPRITE MULTI-COLOUR ONE";A3:IFA3<0ORA3
>15THENPRINT"[2CU]":GOTO3042
3044 INPUT "SPRITE ORDINARY COLOUR";A4:IFA4<0ORA4>
15THENPRINT"[2CU]":GOTO3044
3045 CS=A4
3046 GOSUB 4000
3048 POKE V+37,A2:POKE V+38,A3:POKE V+39,A4
3050 PRINT"[CLR,YEL]":POKE 53281,0:PRINT"[HOME]MOV
E THE CURSOR AROUND THE ";
3051 PRINT"SCREEN WITH THE CURSOR";
3052 PRINT" KEYS. DITTO FOR THE SPRITE USING MEN
U OPTION M.
3053 PRINT"[CDJ]WHEN SATISFIED, TURN IT INTO DATA

```

```

STATEMENTS BY USING OPTION B."
3054 PRINT"THESE ARE LISTED AS LINES 30011-30020."
3055 PRINT"[CD]THE REST OF THE INSTRUCTIONS ARE
      DISPLAYED ON THE SCREEN."
3056 PRINT"[2CD]APART FROM...!  PRESS HOME TO MOVE
      THE CURSOR BACK TO THE TOP LEFT";
3057 PRINT" , AND          CLR/HOME TO ERASE THE SPRI
      TE COMPLETELY."
3058 PRINT"[CD]ALSO, WHEN CHANGING COLOURS, PRESSI
      NG      1  GIVES YOU MULTI-COLOUR ";
3059 PRINT"ZERO, '2'      GIVES YOU MULTI-COLOUR ONE
      ,AND '3' GIVESTHE ORDINARY ";
3060 PRINT"SPRITE COLOUR."
3065 GOSUB 4000
3070 RETURN
4000 PRINT"[2CD]PRESS 'SPACE' TO CONTINUE"
4002 GETSP$:IFSP$<>" "THEN4002
4004 RETURN
29997 REM
29998 REM SPRITE DATA STORED FROM HERE
29999 REM
30011 DATA 13 ,1, 1 , 9 , 12
30012 DATA0,2,128,0,2,128,0
30013 DATA2,128,0,2,128,0,2
30014 DATA128,10,170,128,8,0,0
30015 DATA8,0,0,12,0,0,12
30016 DATA68,0,29,1,0,79,16
30017 DATA64,31,192,16,7,255,244
30018 DATA1,255,253,0,127,244,0
30019 DATA31,208,0,7,64,0,1
30020 DATA0,0,0,0,0,0,0
30021 DATA -1

```

## Notes

A few graphics characters to watch out for here, particularly in lines 180, 182 and 183.

Here we've used CBMP to represent pressing the Commodore logo key and the P key together, and likewise for CBMN, CBMH and CBMY. If you don't get a box appearing on the screen, you've gone wrong somewhere!

There are no other graphic characters used, but you might get problems with lines 3023 to 3038, which are meant to draw up a display of all the available colours on the screen. You'll just have to be careful what you press.

Finally our old friend is at it again, i.e. the up-arrow key has once again appeared as a chinaman's hat in a few places, in lines 626, 808 and 809.

## **To conclude**

I hope you'll find some or all of the above listings useful. They should certainly help towards a better understanding of how sprites work and how they can be manipulated on the Commodore 64.

Before going on to user defined graphics and high resolution plotting, it would help if we actually knew what we were going to be talking about, so we'll conclude chapter four with a fairly detailed look at the graphics chip that does it all, namely the 6566 Vic chip.

## **6566 Video interface chip**

The 6566 is a multi-purpose colour video controller device, capable of being used in quality arcade game terminals, which we have control over in the Commodore 64. It has 47 control registers, which are accessed by any 6502 compatible 8 bit microprocessor, in this case the 6510. In this section we'll be taking a detailed look at its various operation modes and the graphics options it gives us.

### **Character display mode**

In this particular mode, the 6566 fetches character pointers from the video matrix area of memory, and translates that into character dot addresses in the 2K character base of memory. The video matrix consists of 1,000 locations in memory, each containing an 8 bit character pointer.

The location of this video matrix in memory is defined by VM13-VM10 in register 24 (\$19), which are used as the four most significant bytes of the video matrix address. The lower order 10 bits are provided by an internal counter, in VC3-VC0, which steps through each of the 1000 character locations.

The 6566 has some 14 address outputs, as follows:

## Character Pointer Address

A13	A12	A11	A10	A09	A08	.....	A00
<hr/>							
VM13	VM12	VM11	VM10	VC9	VC8	.....	VC0

The 8 bit character pointer permits up to 256 character definitions to be available simultaneously. In other words, under normal operating conditions, we are capable of displaying up to 256 different characters on the screen at once.

Each character consists of an 8 by 8 bit dot matrix, and is stored in the character base as eight consecutive bytes.

The location of this character base is defined by CB13-CB10, which are also installed in register 24. These are used for the three most significant bits of the character base address.

The 11 lower order addresses are formed by the 8 bit character pointer from the video matrix (D7-D0), which selects a particular character, and a 3 bit raster counter (RC2-RC0), which selects one of the eight character bytes.

The resulting characters are formatted onto the screen in 40 columns of 25 rows: a total of 1000 screen locations, as we saw above.

In addition to this 8 bit character pointer, there is a 4 bit colour nibble (simply another word for half a byte, you may recall) associated with each video matrix location. The video matrix must be 12 bits wide.

The colour nibble defines one of the 16 available character colours for each character.

The character data address table looks like this:

## Character Data Address

A13	A12	A11	A10	A09	...	A03	A02	A01	A00
<hr/>									
CB13	CB12	CB11	D7	D6	...	D0	RC2	RC1	RC0

## Standard character mode

In standard character mode, the 8 sequential bytes from the character base are displayed directly on the 8 lines in each character space.

A '0' bit causes the background colour £0 (from register 33, or \$21) to be displayed, while the colour selected by the colour nibble, known as the foreground colour, is displayed for a bit that is set to a '1'. These colour codes have already been given in chapter two, and are simply the ordinary 16 colours available to us.

In other words, each character has a unique colour determined by the 4 bit colour nibble, and all characters must share the same background colour, i.e. the screen background colour.

One of our earlier programs illustrated this, when displaying a collection of randomly coloured spaces on the screen.

To illustrate the point further, it would be possible to change the reverse space printed to be a reverse letter of the alphabet, or indeed any one of the characters available.

Function	Character Bit	Colour Displayed
Background	0	Background Colour £0
Foreground	1	Colour chosen by colour nibble

## Multi-colour character mode

Multi-colour mode gives us a much greater flexibility in choosing our displays, as it allows up to four different colours to be printed within each character space, but our character resolution is now halved in the horizontal direction.

The multi-colour mode is selected by setting the MCM bit in register 23, or \$16, to a '1', which causes all the dot data stored in the character base to be interpreted totally differently.

If the most significant bit of the colour nibble is a '0', then the character will be displayed as described above, in standard character mode.



When it is set to a '0', it is displayed in multi-colour mode as shown below.

Thus the two different modes can be displayed on the same screen, but it is not possible to use any of the colours other than the first 8.

Function	Character Bit	Colour Displayed
Background	00	Background Colour £0
Background	01	Background Colour £1
Foreground	10	Background Colour £2
Foreground	11	Colour from 3 LSB of colour nibble

Since we now require two bits to specify the colour of a dot, each character space is reduced to being 4 pixels by 8 pixels, with each horizontal 'pixel' being the equivalent of two pixels in ordinary mode, in order to allow us to select the colours as indicated. Still, we can now have two background and two foreground colours per character space.

### Extended colour mode

Extended colour mode allows the selection of background colours for each character space within the normal 8 pixel by 8 pixel resolution. Thus greater flexibility of colour choice is given than in standard character mode, without the loss of resolution given by multi-colour mode.

However, extended colour mode and multi-colour mode cannot be used at the same time.

This mode is selected by setting the ECM bit of register 17, or \$11, to a '1'. The character dot data is displayed exactly as in standard character mode, in that the foreground colour as determined by the colour nibble is displayed for every character bit set to a '1', but the two most significant bits of the character pointer are used to select the background colour for each character space as follows:

Character Pointer	Background Colour Displayed for each nibble
MSB Pair	
00	Background Colour £0
01	Background Colour £1
10	Background Colour £2
11	Background Colour £3

Since the two most significant bits of the character pointers are used for colour definition, this means that only 64 characters can be displayed on the screen in extended colour mode. As the 6566 forces CB10 and CB9 to be a '0' regardless of anything else going on, we are limited still further to the first 64 characters. However, these can be either the first 64 characters of ROM, or of your own character set. We'll be looking at this in the next chapter.

This mode allows us to choose any one of 16 foreground colours, and one of four background colours, for each character space.

## Bit map mode

The most powerful of all the graphical modes on the 64. It is also the slowest to operate, and is usually handled from machine code rather than the much slower Basic.

Needless to say, in this mode everything is handled totally differently from all the other modes, and it works in the following way:

In bit map mode the 6566 fetches data from an 8000 byte block of memory, and displays it on the screen in a one-to-one ratio. In other words, each bit of those 8000 bytes relates exactly to a bit as it appears on the screen.

This gives us a maximum resolution of 320 pixels by 200 pixels, or 64000 pixels, or bits. Each bit being one-eighth of a byte, this gives us our (64000/8) 8000 bytes of memory required.

Bit map mode is selected by setting the BMM bit in register 17, or \$11, to a '1'.

The video matrix is still accessed as before, but the data contained there is no longer interpreted as character pointers, but instead it is read as colour data.

The video matrix counter is then also used as an address to fetch the dot data for display from the 8000 byte display base.

The display base address is made up like this:

A13	A12	A11	...	A03	A02	A01	A00
<hr/>							
CB13	VC9	VC8	...	VC0	RC2	RC1	RC0

Where VC denotes the video matrix counter output, RC the 3 bit raster line counter, and CB comes from register 24. The video counter goes through the same 40 locations for eight raster lines, going on to the next 40 locations every 8 lines, while the raster counter is incremented for each horizontal line on the screen (otherwise known as a raster line).

Because of this, each block of 8 sequential memory locations in the 8000 byte base is formatted as an 8 pixel by 8 pixel block on the screen. The first byte is the top line, the second byte the second line, and so on.

### Standard bit map mode

When standard bit map mode is initialised, the colour information comes only from the data stored in the video matrix, and anything that the colour nibble tries to do is totally ignored.

The 8 bits are divided into two 4 bit nibbles, which allows two colours to be independently selected for each 8 pixel by 8 pixel block. When a bit in the display memory is set to a '0' the colour of the output dot is set by the least significant, or lower, nibble, and when it is set to a '1' the colour is selected by the most significant bit, or highest nibble. Thus:

Bit	Colour Displayed
0	Lower nibble of video matrix pointer
1	Higher nibble of video matrix pointer

### Multi-colour bit map mode

Multi-colour bit map mode is selected by setting the MCM bit in register 22, or \$16, to a '1', in addition to setting the BMM bit in register 17.

This mode uses the same 8000 byte block of memory to display its

characters, but this time the data is handled in a totally different way (as you might expect!), like this:

Bit Pair	Display Colour
00	Background Colour £0-register 33
01	Higher nibble of video matrix pointer
10	Lower nibble of video matrix pointer
11	Video matrix colour nibble.

This time we are using the colour nibble, along with the video matrix pointer and the background colour £0 from register 33. Thus we can have three separately chosen colours in each 8 pixel by 8 pixel block, along with one standard background colour.

However, due to the bit pairing to get the colour information, each horizontal pixel is the equivalent of two pixels in ordinary mode, and so our maximum resolution is halved to become 160 dots by 200.

## Sprites

We've already seen how sprites can be displayed and formed, now let's get a little more technical and go into further detail on how sprites behave in the way that they do.

### Sprites on and off

Each sprite can be turned on or off independently of any other sprites that happen to be around, and they are turned on by selecting the corresponding enable bit in register 21 to be a '1'. If this bit is set to be '0', nothing will happen to that particular sprite.

The SID chip memory map was explained in some detail in the chapter on sprites and high resolution graphics.

It sounds wonderful to talk about disabling a sprite - kicking sand in its face - but as we've seen this is simply done by setting the appropriate bit in register 21 to a zero.

### Positioning a sprite

Each sprite is positioned according to X and Y co-ordinates, on a 320 by 200 scale respectively. However, not all these locations can be seen

on the screen, which allows you smooth scrolling off the screen in both horizontal and vertical directions.

For practical purposes, the display should be confined to a vertical scale of 50 to 200, and a horizontal scale of 25 to 315.

The number put into the X and Y co-ordinate registers determines where the sprite will appear on the screen. We gave the memory map for these registers in the earlier part of this chapter.

One other register must be considered, register 16, or \$10. This must be set to a '1' if the X position exceeds 255 (and X re-set to zero), and back to a '0' (and X to 255) again when going the other way. This allows a sprite to travel across the full width of the screen without coming to a sudden halt somewhere near the right-hand side.

### Colouring a sprite

Each sprite has a separate 4 bit register to determine its colour, and as usual there are two different colour modes, known as standard and multi-colour.

In standard mode, a '0' bit of sprite data allows the background colour to show through, and this is referred to as being 'transparent'.

If the bit is set to a '1', then the sprite colour is shown as dictated by the corresponding sprite colour register: see earlier memory map.

But each sprite, regardless of its neighbours, can be selected as a multi-colour sprite by setting the MCS bits in the sprite multi-colour register 29, or \$1C.

When this bit is set to a '1', the sprite will be displayed as a multi-colour sprite, with the colour coming in as follows:

Bit Pair	Colour Displayed
00	Transparent
01	Sprite Multi-colour £0-register 37
10	Sprite Colour-registers 39 through 46
11	Sprite Multi-colour £1-register 38

As we now require two bits to define the colour of each sprite, the resolution of that sprite is halved in the horizontal direction, as each

'pixel' becomes the equivalent of two pixels in standard mode.

It does give us control over three colours per sprite, plus one background colour, but the multi-colours must be the same for all sprites in multi-colour mode.

## Magnifying sprites

Sprites can be doubled in size either horizontally or vertically, or both, and reduced back to normal size again.

Two registers control sprite expansion. If the relevant sprite bit is set to a '1', the sprite is expanded; if set to a '0' it goes back to normal again.

Register	Function
29 (\$1D)	Expand horizontally
23 (\$17)	Expand vertically.

Despite expanding the sprite, we don't get any increase in resolution, as the same 24 pixel by 21 pixel grid is displayed, but expanded in the appropriate direction.

## Priority amongst sprites

This determines which sprite has priority over what, i.e. if sprites pass over each other, or over anything else that happens to be on the screen, register 27 (\$1B) determines what will be displayed.

This can be individually selected for each sprite, by setting the appropriate bit in register 21 to be a '0' or a '1', and functions like this:

Register Bit	Function
0	Non-transparent part of sprite will be displayed
1	Non-transparent part displayed only instead of background colour £0 or multi-colour bit pair 01

Sprites also have a fixed priority amongst themselves, in that sprite

number 0 will always be displayed over sprite number 1, 1 over sprite number 2, and so on.

Sprite to sprite priority is always sorted out before sprite to data-on-screen priority.

## **Sprites colliding**

It is possible to detect two types of sprite collision, sprite to sprite and sprite to anything else on the screen.

A collision between two sprites is said to occur when two non-transparent parts of each sprite want to occupy the same screen area.

When this happens, the appropriate bits in the sprite collision register, register 30 or \$1E, are set to '1' for both sprites.

As more sprites collide, the appropriate bits in register 30 continue to be set, until a read of this collision register, when all the bits are set back to '0' again.

Sprites can even collide off-screen, so watch out!

The second type of collision is between a sprite and anything else on the screen.

When this occurs the appropriate bit for the sprite concerned is set in register 31, or \$1F, to '1', although as before the collision of transparent data does not generate a setting of this register.

The display data from a 01 multi-colour bit pair also does not generate a setting to '1' of this register.

Again, collisions can take place off-screen, and the register is cleared back to zeros again as soon as it is read.

## **Accessing sprites in memory**

We've touched on this one already, but for the sake of complete clarity here we go again.

The data for each sprite is stored in 63 consecutive memory locations,

and some of these we've talked about earlier.

Naturally we have to tell the 6566 where the data for each sprite is stored, and this is done using memory locations 2040 to 2047, the 8 bytes immediately after the screen RAM, and each byte refers to one sprite.

Thus location 2040 refers to sprite 0, 2041 to sprite 1, and so on, up to 2047 which refers to sprite 7.

If a value of 13 is stored in location 2042, it means that the data for sprite 2 is to be found at the 63 memory locations starting at the 13th block of sprite data, which happens to be memory location 832.

The eight-bit sprite pointer, together with the six bits from the sprite byte counter (to address 63 bytes), define the entire 14-bit address field, like this:

A13	A12	A11	A10	A09	A08	A07	A06	A05	...	A00
-----										
SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SC5	...	SC0

Where SP are the sprite pointer bits, and the SCs are the internally generated sprite counter bits. The sprite pointers are read from the video matrix at the end of every raster line.

When the Y position register of a sprite matches the current raster line count, then the sprite data is fetched, with internal counters stepping through the 63 bytes of data, displaying 3 bytes on each raster line.

## Other screen features

As well as all these graphical and sprite features, the 6566 is capable of much more, as we shall see.

### Screen blanking

The display can be blanked off by setting the DEN bit of register 17, or \$11, to a zero. POKE 53265,11 achieves this.

When we blank the display area, the entire screen is filled with the exterior colour as set in register 32, or \$20. This allows us to perform



full processor utilisation of the system bus, or in other words access things like cassette decks, Vic disk drives, and so on.

Sprites, however, unless specifically disabled, continue to shine through.

To get the screen back, the DEN bit must be set to a '1' again, and POKE 53265,27 achieves this.

### Selecting rows and columns

As we've seen, we normally get a 40 column by 25 row screen, but for some purposes it would be desirable to change this. For instance, to enable smooth scrolling of the screen.

This is achieved by altering the RSEL bit in register 17, or \$11, and the CSEL bit in register 22, or \$16, and works like this:

<u>RSEL</u>	<u>Number of Rows</u>	<u>CSEL</u>	<u>Number of Columns</u>
0	24	0	38
1	25	1	40

This effectively moves the border over on to the screen area, but leaves the characters previously displayed there still intact, but no longer visible.

As an example, POKE 53265,19, loses the top half of the top line of the screen, and the bottom half of the bottom line of the screen. POKE 53265,27 gets us back to normal again.

### Scrolling the screen

The entire screen display can be scrolled either horizontally or vertically, one pixel at a time, up to a maximum of one character space. Using this in conjunction with the screen window (screen display minus border) facilities mentioned previously, enables us to produce smooth scrolling of the display area, while updating the system memory only when a new character row or column is required.

This method is also used to centre a fixed display within the screen window.

Bits	Register	Function
X2,X1,X0	22 (\$16)	Horizontal Position
Y2,Y1,Y0	17 (\$11)	Vertical Position

## Light pens

The light pen input stores the current screen position in two registers, labelled LPX and LPY.

The X position is stored in register 19, or \$13, and will contain the 8 most significant bytes of the X position at the time of detection.

As the X position is defined by a 512 (9 bit) state counter, resolution to two pixels is provided.

The Y position, stored in register 20, or \$14, allows us a single raster resolution on the screen display, or down to one pixel.

This light pen input may be triggered only once per frame, or screen scan, and subsequent triggers with that frame will have no effect.

## Raster register

It's always tempting to make comments about people wearing red, green and yellow hats, but instead this is a dual function register.

Reading the raster register 18, or \$12, returns the lower 8 bits of the current raster position. The higher 8 bits are stored in register 17, or \$11.

The visible display window is from raster 51 to raster 251, or \$033 to \$0FB.

A write to the raster bits, including RC3, is stored for use in an internal raster compare, and when the current raster matches this written value, the raster interrupt latch is set.

## Interrupt registers

The interrupt register shows the status of the four sources of interrupt, which are:

Latch Bit	Enable Bit	When Set
IRST	ERST	Raster Count = Stored Raster Count
ISDC	ESDC	Sprite collide with data on screen
ISSC	ESSC	Sprite collide with another sprite
ILP	ELP	Negative transition of LP input (once per frame)
IRQ		

In order for the interrupt request to set the IRQ output to '0', the corresponding interrupt enable bit in register 26, or \$1A, must be set to a '1'.

Again, once an interrupt latch has been set, it may only be cleared by writing a '1' to the appropriate latch in the interrupt register.

## Dynamic screen refresh

Five 8 bit row addresses are refreshed every raster line, and this guarantees a maximum delay of 2.02 ms between the refresh of any single row address in a 129 address system, or 3.66 ms in a 256 address system.

## Reset

The reset bit RES in register 22, or \$16, is not used in the normal mode of operation.

Thus it is normally set low, and setting it high suspends the entire operation of the 6566!

To be used with caution.

## Theory of operation

The 6566 interacts with everything else on board the 64 in a special way.

The 6510 requires access to and from the system buses only during that portion of its cycle known as phase 2, when the clock is set high.

The 6566 takes advantage of this system, and therefore only accesses

memory during phase 1, or when the clock is set low.

Therefore, such operations as getting character data, or refreshing the screen, or anything else that the 6566 handles are totally transparent to the 6510, and thus don't reduce the speed of processor operation. The 6566 itself provides the various interface control signals necessary to perform and maintain this kind of bus sharing.

The 6566 also provides the signal to enable address control, used to disable the 6510 address bus drivers (I nobly resist all comments here!), thus allowing the 6566 to access the address bus for itself. Address Enable Control is active (set low) during phase 1 of the clock cycle, so that again the 6510 is not affected in its speed of operation.

However, because of all this all memory accesses must be completed in at most half a cycle, or 500ns, as the 6566 provides a 1MHz clock.

This could become a problem, since some of the operations of the 6566 require much longer (relatively) than a mere half a cycle. In particular, sprite generation requires that the 6566 also grabs a slice of the action during phase 2, which means that the 6510 must itself be disabled somehow.

This is achieved with the BA, or Bus Active, signal, which is connected to the ROY input of the 6510.

This is normally set high, but can be set low to indicate that the 6566 wants to do some processing during phase 2. In all, the 6566 has three phase 2 times after BA has been set low in order to complete all its data access.

On the fourth phase 2 after BA being set low, the Address Enable Control remains low until the 6566 has finished.

More manipulation must take place during the fetching of the character pointers, which takes some 40 consecutive phase 2 accesses to fetch the video matrix pointers.

Sprites, as we've said, also require more than one phase 2 access, and in fact require four accesses in total, as follows:

Phase	Data	Condition
1	Sprite Pointer	Every Raster Scan
2	Sprite Byte 1	Each raster while sprite is displayed.
1	Sprite Byte 2	As above
2	Sprite Byte 3	As above

Thus sprite pointers are fetched every other phase 1 at the end of every raster line.

All this bus control is handled internally by the 6566 itself, thank goodness.

## Memory interfacing

The 6566 has thirteen fully decoded addresses for direct connection to the system address bus, and can be accessed in the same way as any other peripheral device.

The following 6510 interface signals are provided:

### Data Bus DB7 - DB0

These 8 data bus pins combine to form the bi-directional data port, which can only be accessed while the Address Enable Control and Phase 0 are high, and chip select is low.

### Chip Select CS

This is brought low to enable access to the device registers in conjunction with the address and Read Write pins. It is only recognised as being low when Address Enable Control and Phase 0 are high.

### Read Write R/W

This is used to determine the direction of data transfer on the data bus, in conjunction with CS. When R/W is high, data is transferred from the selected register to the data bus output, and when it is low data presented on the data pins is loaded into the chosen register.

### Address Bus A05-A00

These lower six pins are bi-directional, and are used as inputs during

a processor read or write to the video device. The data on the address inputs selects the register for read or write as defined in the register map.

### **Clock Out PH0**

The clock output, or phase 0, is the 1 MHz clock used as the 6510 processor phase 0 in. All system bus activity is referenced to this clock, the frequency of which is generated by dividing the 8MHz video input clock by 8.

### **Interrupts IRQ**

The interrupt output is brought low when an enabled source of interrupt occurs within the device. It requires an external pull-up register.

### **Video interface**

The output signal from the 6566 consists of two signals, which need to be mixed together.

SYNC/LUM contains all the video data, and requires an external pull-up of 500 ohms.

The Colour output, containing all the colour information for screen display, is terminated with 1,000 ohms to ground.

These two signals are then mixed before being fed through to your television set.

## **Conclusion**

That concludes our look at the capabilities of the 6566 video interface chip. Now that we know how it all works, let's start putting our knowledge into action, and take a look at the art of producing user-defined graphics.

# 5

## User-Defined Graphics

### Introduction

With the Commodore 64 you are given a full complement of characters: numeric characters, alphabetic characters, and a whole host of other graphical symbols.

Using these it is possible to create some very sophisticated displays indeed, and remembering that we also have control over colour and whether or not to print out a character in reverse field, in many instances it is only our imagination that lets us down when it comes to producing exciting screen output.

As if that wasn't enough, we can also display in a variety of different graphical modes, as was seen at the end of the last chapter. Whether we want to go into ordinary high resolution mode, multi-colour mode or extended colour mode, the user is given a great deal of choice on how to display images on the screen.

Finally, of course, we also have sprites. Again we are presented with a choice of different modes when displaying these, and again we can use ordinary sprites or multi-colour sprites. Alas, there is no such thing as an extended colour sprite!

But despite all that there are still times when we wish for a few more different characters from the keyboard. The existing graphics set, albeit a most useful one, is not without its limitations. Most of the characters are straight lines, and the few that do depart from this standard image are of little use except when writing card playing games.

There are times when we are using the existing character set, and we want to add a couple of characters to it. For example, Commodore provide us with a left arrow key and an up arrow key, but not a down one or one pointing to the right. It would be nice, on occasions, to be able to display such characters.

How do we do this ? The answer of course is that we define our own.

## How characters are made up

We know that all the characters available directly from the keyboard are made up of an eight pixel by eight pixel grid, with suitable bits turned on or off depending on what the character is going to look like.

For example, the letter A is made up in the following manner:

```
ABCDEFGH
. . . * . . .
. . * * * . .
. * * . * * .
. * * * * .
. * * . * * .
. * * . * * .
. * * . * * .
. * * . * * .
. . . . .
```

And the '@' symbol looks something like this:

```
ABCDEFGH
. . * * * . .
. * * . * * .
. * * . * * .
. * * . * * .
. * * . * * .
. * * . * * .
. . * * * . .
. . . . .
```

Don't worry about all the letters on the top of each grid, we'll come to those later.

Each character, then, is made up on an eight by eight pixel grid, like the two shown above. You'll note that the alphabetic characters don't extend to the edges of the grid on either the left hand side or the right, nor do they reach the bottom of the grid.

This is true for all the alphabetic characters, and it is done to prevent them merging together as you type. Imagine trying to unravel a program listing where all the characters merged into each other.



On the other hand, the graphical characters (on the whole) do extend to the edges of the grid. This is so that they can be joined up, and thus used to create continuous lines, pictures, and so on.

What we're going to do is to remove some of the existing characters from the Commodore 64's character set, and replace them with some of our own.

Which keys can we choose?

## Re-defining keys

The answer, perhaps not surprisingly, is that we can choose any keys that we feel like.

There are limitations, of course, but by and large we are not too restricted when it comes to re-defining characters. However, most of the time we would not want to re-define the entire keyboard, but would like to keep some of the existing keys. For example, you might like to keep all the alphabetic ones, and the numeric ones, and get rid of everything else.

Whichever you choose is entirely up to you, but first you need to know where everything is stored in the computer's memory.

Normally, unless we tell it otherwise, the 64 will get its character information (that is, the information on how each character is made up), from the character generator ROM, which sits in memory from locations 53248 to 57344, a total of 4K of memory divided up into 8 blocks of 512 bytes each.

Each 512 bytes, remembering that every character takes up 8 bytes of storage space, thus stores information on 64 characters.

In order, these 8 blocks are laid out as follows:

Block	Address		Description
	Decimal	Hex	
0	53248	\$D000	Upper case characters
0	53760	\$D200	Graphics characters

0	54272	\$D400	Reverse case upper case characters
0	54784	\$D600	Reverse case graphics characters
1	55296	\$D800	Lower case characters
1	55808	\$DA00	Upper case & graphics
1	56320	\$DC00	Reverse case lower case characters
1	56832	\$DE00	Reverse upper case and graphics characters

Some of those numbers may seem a little bit familiar, but don't worry. In the 64, nothing ever occupies the same place at the same time, it all gets switched around to cope with everything.

### Choosing characters

To decide which set of characters we're looking at at any one time, we obviously have to point the Vic chip in the right direction.

This is relatively easy for the user to do, but gives the 64 a few problems (thankfully, they're transparent to the user), since the Vic chip which controls all the graphics is only capable of looking at 16K at a time.

Being an 8 bit chip it should be capable of addressing the full 64K all at the same time, but Commodore decided that this was not to be, and instead of the usual number of 16 address lines coming from the chip, they give us just 14. Thus we come down from the giddy heights of 64K to a mere 16K.

To select which block of 16K we want to look at, we have to alter CIA£2 (one of the input/output lines).

First of all, we need to know where the four 16K blocks of memory sit within the 64.

## **Blocks of memory**

To swap from one block to another, we need a statement like:

```
POKE56578,PEEK(56578)OR3
```

This sets bits 0 and 1 of port A of the second 6526 chip to zero.

Then, we need to:

```
POKE56576,(PEEK(56576)AND252)ORA
```

which actually allows us to swap from block to block.

A takes on the following values:

A = 0 - looking at block 0, starting at location  
49152, or \$C000-\$FFFF

A = 1 - looking at block 1, starting at location  
32768, or \$8000-\$BFFF

A = 2 - looking at block 2, starting at location  
16384, or \$4000-\$7FFF

A = 3 - looking at block 3, starting at location 0  
or \$0000-\$3FFF

At power up we are always addressing block 0. If the Vic chip is looking at either block 0 or block 3, then the character ROM is automatically switched into the Vic chip's memory space, starting at location \$1000.

If you want to move the location of the character ROM around yourself, you'll need to:

```
POKE 53272,(PEEK(53272)AND240)ORA
```

where A, reasonably enough, determines where the character ROM is going to sit. If A has a value of 0, it will start at memory location zero, a value of 2 will put it at location 2048 onwards, and so on in steps of 2048.

Well, that's been a lot of theory, let's start putting it into practice!

## Altering character sets

You'll recall how characters are stored on the 8 by 8 pixel grid:

```
ABCDEFGH
. . * * * . .
. * * . . * * .
. * * . * * * .
. * * . * * * .
. * * . * * * .
. * * . . . . .
. * * . . . * .
. . * * * * . .
. . . . . . . .
```

As is usual with bits and bytes, we're going to give a series of values to the letters A to H on the grid, and these are the usual values of 128,64,32,16,8,4,2 and 1.

For each row of the character '@', in order to get the correct data to be stored in the computer, we just need to add up the relevant numbers for all the bits that are turned on in that row.

So, as far as the '@' symbol is concerned, the first row of bits gives us a value of 32 plus 16 plus 8 plus 4, or a total of 60. Continuing on down the character, the seventh row has the same value, and the final row of course has a value of 0.

We're going to use exactly the same principle when re-defining our own characters, and just as an example we'll define a little man.

Taking our usual grid:

A	B	C	D	E	F	G	H
128	64	32	16	8	4	2	1

We get this value by looking at each row in turn, and seeing which bits are turned on and which are turned off. For the first row, you'll see that bits A,B and C are turned off, bits D and E are turned on, and bits F, G and H are turned off.

The next line is exactly the same, but the third line now turns on bits C and F as well as bits D and E, while the others remain off. Again adding the values up gives us a total of 32 plus 16 plus 8 plus 4, or 60. This is the third entry in our data statement.

Taking each line in turn, we arrive at the complete list of values: 24,24,60,90,153,36,36,36. We will use these figures to instruct the computer which bits to turn on and which to turn off for our character. This same character, incidentally, appears in the final program in this chapter.

So, having designed everything on paper, and worked out all the necessary values for each character, it's back to the 64 again.

Just as an example, we're going to keep all the upper case alphabetic and numeric characters, as well as the standard graphic set, and re-define all the reverse image characters. Thus we'll hang on to the first 128 symbols in the character set, and re-define the last 128 symbols.

### **Bringing in the ROM**

Since the character ROM is just that, Read Only Memory, there is no way that we could ordinarily alter it. However, this doesn't prevent us from bringing all this ROM into RAM, and doing what we like with it.

As we've seen, the upper case and graphics characters are normally stored in locations 53248 to 54271. Unfortunately, because of the way the 64 was designed, this puts it between the input/output ROM sitting on top of it, and the ordinary RAM sitting underneath it. So our first task is to remove the input/output ROM, and this is done with a simple POKE:

**POKE 1,51**

However, due to the constantly refreshed interrupts, we must do a little more than this before we can copy in any of the character ROM, since the interrupts will, quite reasonably, expect all that input/output still to be in place. When it finds that it isn't, the machine would simply crash and we'd have to start again. So, to switch off all the interrupts we need to issue the following command:

**POKE 56333,127**

This must be done before removing the input/output.

Now we are in a position where we can copy the first 1,024 bytes of character ROM, since this is where the characters that we want to keep are situated, into the computer's RAM, and this is achieved by:

```
FORI = 0TO1023
POKE 53248 + I, PEEK(53248 + I)
NEXT I
```

### Storing characters

Having got everything into RAM, the locations beyond 54271 are ready to store our own characters. Why 54271? Well, we've just copied in 128 characters, each of which occupy 8 bytes in memory, 8 times 128 is 1024, and since we started at location 53248 we must move another 1024 bytes on, which takes us to memory location 54272 where we can start storing our own characters.

This is done quite simply by reading in data, and then storing it at the appropriate locations, like this:

```
FORI = 0TO7
READA
POKE 54272 + I, A
NEXT
.
.
.
.
.
DATA 24,24,60,90,153,36,36,36
```

Obviously on most occasions you would wish to define more than one character, and so there would be more data statements, and the FOR ... NEXT loop would also have to be adjusted accordingly.

Having read all the data in, we have to restore the input/output to normal again, switch all the interrupts back on, and tell the Vic chip where its character ROM has gone to (or in this case, character RAM!).

This is achieved with the following set of statements:

```
POKE 1,55
POKE 56333,129
POKE 648,196
POKE 56576,4
POKE 53272,21
```

## The complete program

And that gives us our complete program. Putting it all together, you might end up with something like:

```
10 POKE 56333,127
20 POKE 1,51
30 FORI=0TO1023
40 POKE 53248+I,PEEK(53248+I)
50 NEXTI
60 FORI=0TO7
70 READA
80 POKE 54272+I,A
90 NEXT
100 POKE 1,55
110 POKE 56333,129
120 POKE 648,196
130 POKE 56576.4
140 POKE 53272,21
1000 DATA 24.24,60,90.153,36,36,36
```

The character that you have just created can now be displayed in a variety of ways. Most simply, if you switch reverse field on and press the '@' key, instead of a reverse field '@' you will now get a little man. By defining another character to be a little man with his arms raised and then swopping from one character to another, as has been done in the final program in this chapter, you would get an animated effect.

The other way to see your character would be to POKE it onto the screen. For instance, to display the man in the top left-hand corner of the screen, you would:

**POKE 50176,128**

Always remembering to alter the colour as well, with a POKE 55296,1 or whatever.

**POKE 50176?!** Yes, there are a number of side effects to re-defining characters in this way.

First of all, the memory reserved for the screen now starts at location 50176, although it still of course occupies 1000 bytes, one for each screen location. Colour memory, as always, remains unaffected by



all this, and still starts at location 55296.

Our character base, as we've seen, now starts at location 53248. This leaves enough room to re-define 128 characters and store them away, while still keeping all the normal characters. Attempting to go into lower case produces some very strange results indeed!

### **What happens to sprites?**

Finally, we now have to be a bit more careful when designing and displaying sprites. Normally we would find out where the sprite data was stored by referring to memory locations 2040 to 2047, the eight after the memory set aside for the screen RAM. Well, just as the screen RAM has moved up to start at location 50176, so the sprite data pointers have gone with it, and they now live at locations 51192 to 51199.

To find out where to store your sprite data, use the formula:

$$49012 + 74 * A$$

where A is the number of the data block you wish to point the sprite to. For example, if you want to store sprite 0 at data block 13, you'll have to POKE 51192,13 and put the data at location  $49012 + 74 * 13$ , or 49974.

As a bonus for having all these new aggravations, sprites have also grown in size! They now need 72 bytes of data for each one instead of the usual 63, since they now occupy a 24 pixel by 24 pixel grid and not the more usual 24 by 21.

Finally, don't try to use run/stop and restore if you want to break into a program. Some very odd things happen.

## **Other graphics modes**

User-defined graphics, or indeed the existing character set, can be displayed in a number of ways. Earlier, we just talked about POKEing characters onto the screen, or getting the user-defined characters by going into reverse field mode.

However, there are another two display modes that we haven't yet looked at, apart from high resolution which comes in the next chapter,

so we'll round off this one by looking at multi-colour mode and extended colour mode.

## **Multi-colour mode**

Multi-colour mode can be used with either high resolution standard mode graphics, or ordinary standard graphics, and it is a way of getting more than just two colours (foreground and background) into each character square.

Normally, when displaying characters on the screen, they can take on one of two colours. Either the screen background colour, or the character colour reserved for that screen position.

Thus we can display many colours on the screen at once, but our control of colour within each character space is limited.

Multi-colour mode allows us to get around this, but with a lowering of the maximum resolution available on the screen. We are limited to a maximum screen resolution of 160 pixels by 200 pixels, or in other words half the normal horizontal resolution.

This is more than compensated for by the additional display facilities gained, and high-res multi-colour mode can produce some staggering results: we'll be seeing this in chapter six.

For now we'll stick to ordinary characters, and look at the registers that have to be altered to get us into this mode.

To turn on multi-colour mode, you'll need to address memory location 53270 thus:

```
POKE 53270,PEEK(53270)OR16
```

This turns on bit 4, and leaves the rest as they are. To turn it off again,

```
POKE 53270,PEEK(53270)AND239
```

This turns off bit 4, and again leaves the rest of the register as it was before we started. Multi-colour mode is actually set on or off for each character space on the screen, so that it is possible to mix multi-colour and hi-res graphics on the same screen, if required.

This is done by choosing your colours carefully. If the number in colour

memory for that space is less than 8, then that space is produced in ordinary hi-res mode. If the number lies between 8 and 15 then we're into multi-colour mode for that particular space. This is simply altering bit 3 of the colour memory (beginning at decimal 55296, and never moving, unlike just about everything else on the 64).

The colours that can be used in multi-colour mode are dictated as follows:

Each dot in a multi-colour space (8 pixels by 8 pixels, but remembering that each horizontal pixel is the equivalent of two normal ones) can take on one of four colours:

The screen colour, from background register number 0

The colour in background register number 1

The colour in background register number 2

The character colour

These memory locations are respectively:

53281 (\$D021) : bit pair 00

53282 (\$D022) : bit pair 01

53283 (\$D023) : bit pair 10

Colour specified by colour memory, starting at 55296: bit pair 11

It is the bit pairing that determines what colour will be displayed in each pixel, and it operates in the same way as multi-colour does with sprites.

Take, for example, our old friend the character '@', made up in normal high-res graphics like this:

```
00111100
01100110
01101110
01101110
01100000
01100010
00111110
00000000
```

In normal mode, wherever there's a 1 the pixel is turned on (character colour), and where there's a zero it's turned off (screen background colour).

In multi-colour mode we use the bits in pairs (hence the halving of horizontal resolution) like this:

00	11	11	00
01	10	01	10
01	10	11	10
01	10	11	10
01	10	00	00
01	10	00	10
00	11	11	10
00	00	00	00

By reference to the above table we can see which colour each one part of the symbol '@' will come out as.

By bearing all this in mind, and utilising some of the character definition techniques as described earlier, some quite stunning displays can be made without resorting to sprites, or indeed high resolution displays.

To give you just a short programming example, which prints a number of different coloured characters onto the screen:

```
10 POKE 53270,PEEK(53270)OR16 : REM TURN ON MULTI-  
COLOUR  
20 POKE 53282,1 : REM WHITE  
30 POKE 53283,6 : REM BLUE  
40 POKE 53281,5 : REM GREEN  
50 PRINT "ABCDEFGH IJ"  
60 FOR I=0 TO 9:POKE 55296+I,2:NEXT
```

Truly horrendous, but at least it illustrates the variety of colours that can be displayed.

To finish with multi-colour mode (for the time being), it is possible to change instantly the colour of every dot drawn in a particular colour. Thus everything drawn in the screen and/or background colours can be changed immediately.

All you have to alter is either the first or the second background colour.

Finally, a quick way of getting into multi-colour mode is to turn it on in the usual way with the POKE 53270 etc., and then change the colour of everything using the logo key. Thus, logo and 9 will give you a brown colour, or multi-colour white.

## **Extended background colour mode**

This is a further extension to multi-colour mode, and gives you the ability to choose both background and foreground colours for each character. You lose a few colours, like all the ones normally accessed with the logo key, but you do get a much greater choice of combinations.

For example, you could display a white character with a blue background on a black screen, or something like that, which gives you the ability to produce some rather interesting three dimensional effects.

It is invoked with the following command:

```
POKE 53265,PEEK(53265)OR64
```

This turns on bit 6, and leaves the rest as they were. To turn it off again:

```
POKE 53265,PEEK(53265)AND191
```

This turns bit 6 off, and again leaves all the other bits unaltered. As usual, there is a trade-off for this, and we are now restricted to using just the first 64 characters of character ROM, or of course the first 64 characters of your own user defined character set.

This is because we need to know which background colour to display everything in, and this is determined by two bits of the character code, like this :

For 0 to 63:	select background colour with location 53281
For 64 to 127:	select background colour with location 53282
For 128 to 191:	select background colour with location 53283
For 192 to 255:	select background colour with location 53284

Choosing screen values greater than 63 will simply reflect back onto the screen the character corresponding to itself in the first 64 screen codes, but in a different background colour.

Thus POKEing a 72 to the screen will still produce a letter H on the screen, but in a different background colour than if you'd just POKEd an 8 onto the screen.

For instance:

```
5 POKE 53265,PEEK(53265)OR64
10 PRINT"[CLR]"
15 FORI=0TO3:POKE 53281+I,INT(RND(.5)*15):NEXTI
25 FORI=0TO255
30 POKE1024+I,I
40 NEXTI
50 GOTO10
```

You may have to play about with screen background colours, depending on what you already had up there!

## Character generator

Now that you've been doing it all the hard way for so long, here's a program that will take the trouble out of re-defining characters, by doing all the hard work for you. All that's required from you is the imagination!

When you run the program at first, don't be alarmed if nothing seems to be happening other than the screen clearing, a few words appearing on the screen, and then absolutely nothing else. This is merely due to the program copying the character ROM into locations 12288 upwards, and as you have seen with the small program given earlier, this does take some time.

When the program does re-appear, you'll be given a grid of characters on the left-hand side of the screen, and a list of options on the right to either create a new character of your own, edit an existing one, or simply quit the program.

Quit simply takes you out of the program and changes the screen and border colours to what they were at power on. Either of the other two options will bring a new display onto the screen.

This consists of an eight by eight grid of dots, with blobs where we would like the pixels to be turned on (if indeed any are turned on at all). The normal cursor controls move your little cursor about the screen, and you can either add or remove dots as you wish. Updating the masterpiece when you're satisfied with it will give you the data for each row of pixels for the new character.

You can either just jot these down, or if you wish the program will turn them into data statements when you press the appropriate key.

The data thus produced can be used with the program given earlier provided that you remove the first number in the data statement. This is just used by this program to keep track of where all the characters are stored in RAM, and need not be a part of any of your own programs.

### **Program notes**

Line 130 : sets screen and border colours.

Line 140 : screen messages.

Lines 150-205 : copy character ROM into RAM.

Lines 210-260 : add new line to Basic program so user doesn't have to sit through a long wait again!

Lines 280-310 : set starting parameters.

Lines 330-340 : set up functions.

Line 350 : print on-screen instructions.

Lines 360-460 : print screen display when editing.

Lines 470-770 : await and act upon user input.

Lines 810-840 : ditto when starting up.

Lines 1000-1170 : set up on-screen instructions and initial display.

Lines 1210-1300 : display on-screen editing options.

Lines 1520-1550 : exit from program.

Lines 1600-1710 : handle update option i.e. display pixel data on screen.

Lines 1810-1910 : selected edit option, display grid and character data.

Lines 2010-2160 : produce data statements from pixel data.

```

130 POKE 53280,9:POKE 53281,0
140 PRINT"[CLR,YEL]          * CHARACTER DESIGNER *
";PRINT"[3CD]PLEASE HANG ON!"
150 POKE 828,0
160 RUN 170
170 CS=12288
175 POKE 56334,PEEK(56334)AND254:POKE 1,PEEK(1)AND-
251
180 FOR I=CS TO CS+2047
190 POKE I,PEEK(53248+I-CS)
200 NEXT I
205 POKE 1,PEEK(1)OR4:POKE 56334,PEEK(56334)OR1
210 PRINT"[CLR]5 RUN 280"
220 PRINT"RUN"
230 POKE 198,3
240 POKE 631,19
250 POKE 632,13
260 POKE 633,13
270 END
280 S=1024:CL=40
290 CS=12288
300 CR=0:LN=30000+PEEK(828)
310 P=24:BG=1:BR=1
320 POKE 53280,9:POKE 53281,0
330 DEFFNA(XX)=S+R*2*CL+2*C:REM SCREEN POKE LOCATI
ON
340 DEFFNB(XX)=8*R+C:REM SCREEN POKE VALUE FOR CHA
R
350 GOTO 1000
360 PRINT"[CLR,YEL]":GOSUB 1200
370 PRINT"[HOME]";:FOR I=0 TO 7
380 PRINT". . . . .":PRINT
390 NEXT I:F=0
400 PRINT"[HOME]":R=0:C=0
410 Z=FNA(0)
420 IF F=0 THEN 460
430 IF Z=ZL THEN 450
440 POKE ZL,IL:ZL=Z:IL=PEEK(ZL)
450 POKE Z+54272,6
460 POKE Z+54272,6
470 GET A$:IF A$="" THEN 470
480 POKE Z+54272,7
490 REM
500 REM CURSOR CONTROL OPTIONS
505 REM
510 IF A$="Q" THEN 1500
520 IF A$="[CR]" AND C=7 THEN C=0:GOTO 410
530 IF A$="[CR]" THEN C=C+1:GOTO 410
540 IF A$="[CL]" AND C=0 THEN C=7:GOTO 410
550 IF A$="[CL]" THEN C=C-1:GOTO 410
560 IF A$="[CD]" AND R=7 THEN R=0:GOTO 410
570 IF A$="[CD]" THEN R=R+1:GOTO 410

```



```

580 IF A$="[CU]" AND R=0 THEN R=7:GOTO 410
590 IF A$="[CU]" THEN R=R-1:GOTO 410
600 IF A$="[HOME]" THEN 400
610 IF F=1 THEN 800
695 REM
700 REM DEFINE NEW CHARACTER OPTIONS
705 REM
710 IF A$="+" THEN POKE Z,81:GOTO 410
720 IF A$="--" THEN POKE Z,46:GOTO 410
730 IF A$="=" THEN 1600
740 IF A$="[CLR]" THEN 370
750 IF A$="R" THEN 1000
760 IF A$="B" THEN 2000
770 GOTO 410
795 REM
800 REM REVIEW CHARACTER SET OPTIONS
805 REM
810 CR=FNB(0)
820 IF A$="N" THEN POKE 53272,21:GOTO 360
830 IF A$="E" THEN POKE 53272,21:F=0:GOTO 1800
840 GOTO 410
995 REM
1000 REM DISPLAY CHARACTER SET OPTIONS
1005 REM
1010 POKE 53272,(PEEK(53272)AND240)+12:R=4:C=0
1020 ZL=FNA(0):IL=32
1030 F=1:PRINT"[CLR]";
1040 PRINT"[YEL]@ A B C D E F G":PRINT
1050 PRINT"H I J K L M N O":PRINT
1060 PRINT"P Q R S T U V W":PRINT
1070 PRINT"X Y Z [ \ ] ^ _":PRINT
1080 PRINT" ! "CHR$(34)" # $ % & '":PRINT
1090 PRINT"( ) * + , - . /":PRINT
1100 PRINT"0 1 2 3 4 5 6 7":PRINT
1110 PRINT"8 9 : ; < = > ?":PRINT
1120 PRINT"[HOME]"SPC(22)"OPTIONS":PRINT
1130 PRINTSPC(22)"[RVS]N[OFF] NEW CHAR":PRINT
1140 PRINTSPC(22)"[RVS]E[OFF] EDIT CHAR":PRINT
1150 PRINTSPC(22)"[RVS]Q[OFF] QUIT"
1160 BC=PEEK(55296)
1170 GOTO 410
1195 REM
1200 REM EDIT OPTIONS
1205 REM
1210 PRINT"[HOME,YEL]"SPC(25)"OPTIONS":PRINT
1220 PRINT
1230 PRINTSPC(P)"[RVS]+[OFF] ADD DOT":PRINT
1240 PRINTSPC(P)"[RVS]-[OFF] ERASE":PRINT
1250 PRINTSPC(P)"[RVS]=[OFF] UPDATE":PRINT
1260 PRINTSPC(P)"[RVS]R[OFF] REVIEW":PRINT
1270 PRINTSPC(P)"[RVS]Q[OFF] QUIT":PRINT
1280 PRINTSPC(P)"[RVS]B[OFF] ADD DATA":PRINT

```

```

1290 PRINTSPC(P+1)"[CU]STATEMENT"
1300 RETURN
1495 REM
1500 REM QUIT
1505 REM
1510 REM
1520 POKE 53272,21
1530 POKE 53281,6:POKE 53280,14
1540 PRINT"[CLR]"
1550 END
1595 REM
1600 REM UPDATE
1605 REM
1610 PRINT"[HOME]":
1620 X=CS+8*CR
1630 FOR R=0 TO 7:SM=0
1640 FOR C=0 TO 7:D=7-C
1650 SM=SM-2^D*(PEEK(FNA(0))=81)
1660 NEXT C
1670 POKE X+R,SM
1680 PRINTSPC(17);SM:PRINT
1690 NEXT R:R=0:C=0
1700 GOTO 410
1795 REM
1800 REM EDIT CHAR
1805 REM
1810 PRINT"[CLR]"
1820 X=CS+8*CR
1830 FOR R=0 TO 7:Y=PEEK(X+R)
1840 FOR C=0 TO 7:Z=FNA(0)
1850 Q=46:Y=Y*2
1860 IF Y>255 THEN Q=81:Y=Y-256
1870 POKE Z,Q:POKE Z+54272,7
1880 NEXT C,R
1890 R=0:C=0
1900 GOSUB 1200
1910 GOTO 410
1995 REM
2000 REM ADD DATA STATEMENTS
2005 REM
2010 X=CS+8*CR
2020 PRINT"[CLR,BCD]"
2030 PRINTLN:"DATA";
2040 PRINTRIGHT$(STR$(X),LEN(STR$(X))-1);
2050 FOR I=X TO X+7
2060 PRINT", ";
2070 PRINTRIGHT$(STR$(PEEK(I)),LEN(STR$(PEEK(I)))-1);
2080 NEXT I
2090 PRINT:PRINT"RUN [HOME]"
2100 POKE 828,PEEK(828)+1
2110 POKE 198,9

```

```
2120 FOR I=0 TO 8
2130 POKE I+631,13
2140 NEXT I
2160 END
```

## Notes

There shouldn't be any problems with this listing, since there are no graphic characters printed to the screen at all, everything is simply POKEd. However, there is one character which you'll have to watch out for, and yes, you guessed it, the strange symbol in line 1650 is indeed meant to be an up-arrow!

## Movemaze

Originally written by Jim Butterfield for one of the ancient Commodore PETs, this program has been revised and updated to take into account some of the facilities of the Commodore 64, and in particular to use a couple of user defined characters.

The game is basically a straightforward maze game, but with one important difference. In this one, the walls move as you walk along through it! You can barge through a wall if you want, but this adds ten points to your score (normally it just goes up by one point for every move that you make), and since the idea is to reach the exit in as few moves as possible this is not a good idea.

You are shown moving through the maze as a little man, and your friend can be seen guiding you to the exit as he jumps up and down and waves his arms about. Both of these characters are, of course, user-defined graphics.

### Program notes

Lines 90-92 : set up screen and border colour as well as sound parameters.

Lines 100-176 : on-screen instructions, and GOSUB to set up user-defined graphics.

Lines 200-290 : set up maze on screen.

Lines 300-330 : setting up parameters.

Lines 340-364 : put characters on screen and await input.

Lines 370-379 : accept or reject input, convert accordingly, and play a note.

Lines 380-390 : checking input.

Lines 400-430 : update progress.

Line 440 : if there isn't a space in front of you there could be trouble!

Lines 450-460 : okay, update position and return

Line 470 : if there isn't a shifted space (exit) in front then you are in trouble!

Lines 480-510 : home and dry!

Lines 520-760 : more housekeeping, plus move a piece of the maze around.

Lines 770-920 : oops! crashed through a wall.

Lines 20000-20130 : character ROM routine.

Lines 30000-30001 : data for new characters.

```
90 POKE 53281,0:POKE 53280,8:PRINT"[YEL]"
92 S=54272:POKE S+24,15:POKE S+5,9:POKE S+6,108:PO
KE S+4,33
100 REM MOVING MAZE DEC/80 JIM BUTTERFIELD
102 REM MODIFIED JAN/84 FOR CBM64 PETE GERRARD
110 PRINT"[CLR,3CR]MOVING MAZE
120 PRINT"[CD] JIM BUTTERFIELD
130 PRINT"[2CD]THE MAZE CHANGES AS YOU MOVE THROUG
H IT TO TRY AND RESCUE YOUR ";
132 PRINT"LITTLE FRIEND BY THE EXIT."
140 PRINT"[2CD] .. YOU MAY PUSH THROUGH A WALL BUT
IT
150 PRINT"WILL COST YOU 10 MOVEMENT POINTS.
160 PRINT"[CD] MOVE WITH KEYS:
170 PRINT"          (UP)
171 PRINT"          I
172 PRINT"      (LEFT) A      D (RIGHT)
173 PRINT"          M
174 PRINT"          (DOWN)
176 PRINT"[2CD]JUST HANG ON A BIT!":GOSUB 20000:PR
```

```

INT"[CLR,CD,YEL]
180 PRINT"[CD] PRESS ANY KEY WHEN READY.[GRN]"
190 GETX$:IFX$=""GOTO190
200 POKES+24,0:PRINT"[RVS,SHIFTN,CLR]";PRINT" ":FO
RJ=0T090:IFPEEK(50176+J)=32THENNEXTJ
210 L=J:L1=(L-10)/2:L$=" "+CHR$(111):R$=CHR$(165):
M$=CHR$(163):FORJ=1T09
220 PRINTL$;:FORK=1TOL-8:PRINTM$;
230 NEXTK:PRINT"[CL]";R$
240 IFJ=9GOTO280
250 PRINT" "CHR$(165);:FORK=1TOL-8:PRINT" ";:NEXTK
:PRINT"[CL]";CHR$(165)
260 L$=" "+CHR$(165):R$=CHR$(165):M$=" "
270 NEXTJ
280 PRINT"[CU,SP]"CHR$(163);:FORK=1TOL-8:PRINTCHR$
(163);:NEXTK:PRINT"[CL]";" "
290 FORV=0T06:FORH=0TOL1-1:H%=V+H:IFINT(H%/2)=H%/2
GOTO320
300 GOSUB520:V%=RND(1)*2
310 GOSUB530
320 NEXTH:NEXTV
330 M=0:V=0:H=-1:GOSUB520
340 C=X+1-L:POKEC,129:POKE 55296+C-50176,7
350 V=6:H=L1:GOSUB520:POKEX-L,96
355 POKE 50176+634,129:POKE 55296+634,1
360 POKE S+24,0:GETX$:IFX$=""GOTO360
362 POKE 50176+634,129-BB:BB=1-BB
364 POKE S+4,33:POKE S+24,15
370 X=ASC(X$)-64:IFX<0ORX>14GOTO360
372 IFX=13THENX=1:POKE S+1,4:POKE S,73:GOTO380
374 IFX=1THENX=3:POKE S+1,5:POKE S,103:GOTO380
376 IFX=9THENX=7:POKE S+1,6:POKE S,108:GOTO380
378 IFX=4THENX=5:POKE S+1,7:POKE S,53:GOTO380
379 GOTO 360
380 V%=X/3:H%=X-V%*3
390 V%=1-V%:H%=H%-1:IFV%*H%<>0GOTO360
400 M=M+1:PRINT"[HOME,YEL]";M;"[CL] MOVES[GRN]"
410 IFV%+H%=0GOTO460
420 M1=C+L*V%+H%
430 M2=C+2*(L*V%+H%)
440 IFPEEK(M1)<>32GOTO470
450 POKEC,32:C=M2:POKEC,129:POKE 55296+C-50176,7
460 GOSUB720:GOTO360
470 IFPEEK(M1)<>96THENGOSUB770:GOTO360
480 POKEC,32:C=M2:POKEC,129:POKE 55296+C-50176,7
490 PRINT"[HOME,YEL]HOME IN";M;:PRINT"MOVES! ANOTH
ER GAME (Y OR N)[GRN]"
492 GETZ$:IFZ$="Y"THEN200
494 IFZ$="N"THENPRINT"[CLR,CD]BYE":FORI=0T024:POKE
S+1,0:NEXT:END
496 GOTO 492
510 PRINT"[CLR]":END

```

```

520 X=(V+2)*L*2+(H+2)*2+50175:RETURN
530 IFV%>0GOTO670
540 GOTO590
550 FORJ=X-L*2TOX+LSTEPL:K=PEEK(J):Z=32:IFK=101GOTO
5580
560 Z=99:IFK=79GOTO580
570 GOTO 360
580 POKEJ,Z:NEXTJ
590 FORJ=X-2TOX+1:K=PEEK(J):Z=99:IFK=32GOTO620
600 Z=79:IFK=101GOTO620
610 GOTO 360
620 POKEJ,Z:NEXTJ:RETURN
630 FORJ=X-2TOX+1:K=PEEK(J):Z=32:IFK=99GOTO660
640 Z=101:IFK=79GOTO660
650 GOTO 360
660 POKEJ,Z:NEXTJ:GOTO670
670 FORJ=X-L*2TOX+LSTEPL
680 K=PEEK(J):Z=101:IFK=32GOTO710
690 Z=79:IFK=99GOTO710
700 GOTO 360
710 POKEJ,Z:NEXTJ:RETURN
720 V=INT(RND(1)*7):H=INT(RND(1)*L1):H%=V+H:IFINT(
H%/2)=H%/2GOTO720
730 GOSUB520:K=PEEK(X)
740 IFK=101GOTO550
750 IFK=99GOTO630
760 GOTO 360
770 M=M+10:Q=M1-50175-4*L-4
772 POKE S+4,129
780 V2=INT(Q/L):H2=Q-V2*L
790 V=INT(V2/2):H=INT(H2/2)
800 IFV=V2/2GOTO840
810 IFH=H2/2GOTO870
820 GOTO 360
830 V=V1:H=H1:GOTO730
840 H=(H2+1)/2:GOSUB910
850 IFETHENH=(H2-1)/2
860 GOTO890
870 V=(V2+1)/2:GOSUB910
880 IFETHENV=(V2-1)/2
890 IFV=>0ANDV<7ANDH=>0ANDH<L1GOTO730
900 RETURN
910 H%=V+H
920 E=(INT(H%/2)=H%/2):RETURN
20000 POKE 56333,127
20010 POKE 1,51
20030 FORX=0TO1023
20040 POKE 53248+X,PEEK(53248+X)
20050 NEXT X
20060 FORX=0TO15
20070 READA:POKE 54272+X,A
20080 NEXTX

```

20090 POKE 1,55  
20100 POKE 56333,129  
20110 POKE 648,196  
20120 POKE 56576,4  
20130 POKE 53272,21  
30000 DATA153,90,60,24,24,36,36,36  
30001 DATA24,24,60,90,153,36,36,36  
30002 RETURN





## 6

# High Resolution Graphics

## Introduction

High resolution plotting is arguably the most powerful of all the graphical features available on the Commodore 64, but it is perhaps not invoked as often as you might imagine, owing to the prodigious demands on the computer's memory.

To fill a full screen with a high resolution display requires some 8K of memory to handle it, since we are now controlling every individual pixel on the screen (at least, in standard high resolution! Multi-colour, as usual, halves our horizontal resolution). Since the screen has 1,000 character locations on it, and each one of those consists of 64 pixels (8 pixels square), we have to look after 64,000 pixels, or 8000 bytes. A lot of memory to control.

However, the rewards justify the efforts involved, even in Basic, although we will be giving a few routines in machine code towards the end of this chapter to speed things up a little.

Even in standard high resolution mode, where each character space is limited to just two different colours (foreground and the usual screen background), some marvellous displays can be very easily generated. Multi-colour mode gives us a more generous four colours per character space, but each pixel in this mode becomes equivalent in size to two pixels in the ordinary mode, so we lose a little on resolution, but nevertheless gain an awful lot on the type of displays that we can achieve.

### Different modes

With ordinary graphical displays we also had a third choice when it came to displaying characters, namely extended colour mode.

However, in high resolution plotting we have no such luxuries, and

we are left with just the two modes already mentioned.

Still, this is more than generous by the standards set by most other home computers, and it is up to 64 users to make the most of it: the hardware code is there for you to use, all we've got to do is provide the software!

## **Getting started**

There are two very important registers when it comes to manipulating everything in high resolution mode, and these are registers 53265 and 53270.

If you glance at the video chip memory map given in chapter four, you'll see that these correspond to locations  $V + 17$  and  $V + 22$ , and that, as well as invoking high resolution, they can also be used to move the screen about, set windows, and so on. We'll be coming to those latter topics later on, but for now we'll stick with straightforward graphics.

As we've seen, it requires some 8K of memory to display anything in high resolution on the screen. This does not mean that the amount of memory that can be stored on the screen has increased to 8K, nor that the screen memory itself has been increased. It has, and always will have, just 1K set aside for it. Rather, each screen location remains as one byte, but to generate the image displayed in it requires 8 bytes.

So what we're doing in this chapter is basically transferring an 8K section of your computer's memory directly onto the screen, and this will determine whether each bit on the screen is to be turned on or off, and in what colour it is to be displayed.

As the only two Basic commands which allow us to transfer memory from one place and store it in another are PEEK and POKE, you begin to get some idea of why all this is so slow. Later on there are some routines for doing this kind of thing in machine code, but we'll save those for a little while. For now, on with:

## **Standard bit mapping**

As with normal character displays on the screen, standard mode gives you fewer colours, but greater resolution: the full 320 by 200, in fact, but only two colours per 8 pixel by 8 pixel square. High resolution and

normal displays actually relate quite closely to each other, but moving from one to the other is not very straightforward. Still, two of the routines at the end of this section do allow you to swop from one type of screen to another virtually instantaneously.

Bit map mode is turned on with the following command:

```
POKE 53265,PEEK(53265) OR32
```

This turns on bit 5 of this register and leaves the rest unaffected. We turn off bit map mode by issuing the following command:

```
POKE 53265,PEEK(53265)AND223
```

This time we're turning bit 5 off, and once again every other bit remains unaffected. This is much to be preferred to the usual method of POKE FRED,BILL, which can sometimes have quite devastating effects!

Obviously we're going to have to get our bit map information from somewhere, and for this we'll have to clear out a section of memory. To start with, we'll wipe out memory location 8192 onwards, so to clear out our 8K we must:

```
FOR I = 8192 TO 8192 + 7999:POKE I,0:NEXT
```

which is not the quickest of processes.

Now we've got to select a few colours, and the colour displayed on the screen in this mode is determined not by the colour memory, but by the actual content of each screen memory location. The value POKEd into a screen location produces the background colour from the lower 4 bits (or lower nibble) of that value, and the pixel colour from the upper four bits (or upper nibble). Thus each screen character space can have two colours in it, and throughout the screen we can use any of the 16 colours.

It might be easier, instead of thinking of upper nibbles and lower nibbles, to picture the byte split up into two separate halves. This:

Bit No.	0	1	2	3	4	5	6	7
Value	1	2	4	8	16	32	64	128

is how we normally would look at it. Now, we have something like:

Bit No.	0	1	2	3	4	5	6	7
Nibble	Lower				Upper			
Value	1	2	4	8	1	2	4	8

Now the upper half can take a value from 0 to 16, and so can the lower half.

## Screen printing

Before printing anything on the screen, we need to tell the screen where our bit map is stored, and this is done with:

```
POKE 53272,PEEK(53272) OR8
```

which puts the bit map at locations 8192 through to 16191 by setting bit 3 to be on. Setting one of the other bits to be on would move the bit map locations correspondingly, although you must remember to wipe out the correct part of memory as well, otherwise ... disaster.

Also, if we don't tell the screen where to go, some very odd things happen! For instance, POKE 53265,59 without having previously set location 53272 will show the top half of the screen containing the bit map from the first 4096 memory locations, and the bottom half from the character generator area. You can actually watch it all change up at the top, as page zero continues to monitor what's happening.

It makes an interesting experiment, if nothing else, and as you hammer away at the keys trying to restore the machine to order, page zero happily carries on its work, oblivious.

Assuming we've done everything correctly, try entering the following line:

```
FORI = 8192 TO 8511 STEP 8: POKE I, 255: NEXT I
```

This will now produce a hi-res line across the top of the screen, given that we're still looking at locations 8192 upwards for our hi-res area.

This is due to the way in which the screen data is interpreted. The 8000 memory locations are not viewed as a continuous set of data. Instead, we have to take the first row of pixels from each of the first 40 bytes to form the top row of our high resolution screen, the second row of pixels forms the second row of the screen and so on, until on the 41st byte it all starts all over again!

This can be confusing at first, but there are ways in which we can calculate the position of any pixel on the screen.

Or again:

```
FOR I = 1024 TO 2023: POKE I, 4: NEXT I
```

will produce a purple hi-res screen.

To discover whether any particular pixel is to be on or off, we'll need to find it on the screen, and the following formula will show the location of any pixel on the screen, assuming we want it to be at X location horizontally, and Y location vertically, with our origin at the top left:

```
R = INT(Y/8)      : find the row
C = INT(X/8)      : find the column position
L = Y AND 7       : the line of that character position
B = 7 - (X AND 7) : the bit of that byte
```

Putting them all together gives us the byte where any pixel with the co-ordinates X,Y is situated:

```
BYTE = 8192 + R*320 + C*8 + L
```

and to turn any X,Y co-ordinate bit on in that 8 by 8 space we:

```
POKE BYTE, PEEK(BYTE) OR (2 to the power B)
```

We'll use some of the theory that we've learnt above to get the 64 to draw an interesting plot on the screen.

```
5 POKE 53272, PEEK(53272) OR 8
7 POKE 53265, PEEK(53265) OR 32
10 FOR I = 8192 TO 8192 + 7999: POKE I, 0: NEXT I
12 FOR I = 1024 TO 2023: POKE I, 1: NEXT I
20 DEF FN A(Y) = 90 * EXP(-Y * Y / 1500)
30 FOR A = 1 TO 2
40 G = 160: K = 60: FOR X = -100 TO 0: L = -60: H = 5 * INT(SQR(10000 - X * X) / 5)
50 FOR Z = H TO H - HSTEP - 5: Y = 25 + FN A(SQR(X * X + Z * Z)) - .6 * Z
60 IF Y > L THEN L = Y
62 C = INT((G + X) / 8): R = INT((K + Y) / 8): L = (K + Y) AND 7
64 BYTE = 8192 + R * 320 + C * 8 + L
66 B = 7 - ((G + X) AND 7)
68 POKE BYTE, PEEK(BYTE) OR 2 ^ B
```

```

70 Q=INT((G-X)/8):S=INT((K+Y)/8):T=(K+Y)AND7
71 BYTE=B192+S*320+Q*8+T
72 D=7-((G-X)AND7)
73 POKE BYTE,PEEK(BYTE)OR2^D
75 NEXTZ,X:GOSUB1690
80 DEFFNA(Y)=38*(SIN(Y/24)+.48*SIN(3*Y/24))+20:NEXT:END

```

Now, I'm not saying that this is the fastest program in the world (you've time to make a cup of tea, smoke a cigarette, make a cup of tea, smoke ... etc.), but IT WORKS!

To plot your own functions in hi-res, or indeed just plotting a couple of individual points, it is simply a matter of working out the X and Y co-ordinates on a scale of 0 to 320 for X and 0 to 200 for Y, remembering that the origin is the top left hand corner of the screen, and then simply using the formula given earlier to get the display happening.

By using functions as we have here, life has been made a little easier. You're welcome to try any function you can think of and see what happens.

## Multi-colour bit mapping

This is similar to multi-colour mode in ordinary graphics, as we are again allowed to have up to four colours per 8 pixel by 8 pixel grid, but we have to suffer a halving of the horizontal resolution available, down to 160 by 200 pixels.

As usual, this is because each horizontal pixel becomes the equivalent of two ordinary pixels, in order to allow us to define these four colours.

Again, we are using an 8K section of memory, and our four colours are chosen from:

Screen background colour, register 53281.

Character screen position, where the upper four bits give us one colour, the lower four another.

Colour memory.

To turn multi-colour bit mapping on, we must:

```
POKE 53265,PEEK(53265)OR32:POKE 53270,PEEK(53270)OR16
```

and to turn it off again:

```
POKE 53265,PEEK(53265)AND223:POKE 53270,PEEK(53270)AND239
```

Referring back to the section on ordinary multi-colour mode in chapter five, the bit pairs are now read as follows:

00 takes on the screen background colour.

01 comes from the upper four bits of screen memory.

10 from the lower four bits.

11 from the colour memory.

Thus using multi-colour mode graphics in high resolution is not markedly different from using it in standard graphics modes.

## **Moving the screen about**

With all these capabilities at your disposal, it is not surprising to learn that graphically the Commodore 64 can do some rather amazing things.

It is possible, for instance, to move the screen either horizontally or vertically in either direction, just one pixel at a time.

This kind of high resolution movement makes a lot of things possible, such as spreadsheet programs, word processors and the like, and it is achieved in the following manner.

The 64 normally displays a screen that is 40 columns across and 25 rows down, but in order to scroll in either direction we can change this into a 38 by 24 display, in order to give the screen information somewhere to go to, and come from.

To go into a 38 column screen display, we must enter:

```
POKE 53270,PEEK(53270)AND247
```

and to get back again we must:

```
POKE 53270,PEEK(53270)OR8
```

To get to a 24 row screen display, we must enter:

```
POKE 53265,PEEK(53265)AND 247
```

and to go back to 25 rows again:

```
POKE 53265,PEEK(53265)OR8
```

While all this is going on, the screen border will expand and shrink again in size accordingly, in order to accommodate the screen manipulation. This does not mean that we are losing any characters simply because the border is now covering them. They're still there, waiting to be read (or rather, PEEKed) by the computer, in order to decide what to do with them.

To scroll horizontally, we must then:

```
POKE 53270,(PEEK(53270)AND248) + X
```

where X is the screen position from 0 to 7. To scroll vertically:

```
POKE 53265,(PEEK(53265)AND248) + Y
```

where Y is the Y position of the screen from 0 to 7.

To see how this works in practice, let's look at a few practical examples.

Values in the range 24 to 31 actually control the vertical position of the characters on the screen, so:

```
FORJ=24TO31:POKE53265,J:NEXTJ
```

will set the screen moving downwards, leaving an empty space near the top. POKE53265,27 to get back to normal.

To illustrate 24 column mode, type POKE 53265,19, which cuts the top and bottom lines in half, and this is the basis of all these scrolling operations.

Switch to a 24 character screen, move everything up slowly, then jump back to a 25 character screen again, and so on.

Finally, to turn the screen off completely, type POKE 53265,11.

POKEing 53265 with 27 always sets everything back to normal again.

There's a lot of material to follow in all of that, and the only way to understand it all is just to play around, taking notes of everything you do and the results that follow.



You can't harm your computer, and Run/Stop and Restore will usually get you out of trouble without too much bother.

To finish off with, and to leave you with a game before we start taking a look at sound, playing around on the 64 one day I made (as everyone does) a typing error. I was trying to get the machine into lower case, but instead of altering register 53273 to 23, I accidentally altered register 53272 to 23.

The result was so interesting that it had to be pursued, and I finally discovered that POKEing 53272 with a 25 gave satisfactory results. This will give you a 40 column by 25 row display all right, but in a totally different character set. The letters manage to look very futuristic, and could save someone a lot of time and memory if they were trying to re-define the entire character set to look like this.

## **The Thinker**

This is a variation on a Mastermind theme, but it also uses that interesting character set that I just mentioned.

The player can choose from having to guess 4, 5 or 6 different colours, and whether or not colours are to be repeated.

Guesses are entered by pressing the relevant key on the keyboard. For instance, if you think the first counter is black, enter your first guess by pressing the key marked BLK. The colours used are black, white, red, purple, green and blue.

The computer will then mark your guess in the traditional way, by displaying a tick for a piece that is the correct colour and in the correct place, or a cross if you've got the right colour but the wrong place. Of course, if your guess is completely wrong in both placing and colour, you won't be told anything.

If you want to give up, you can just press the left arrow key (which is the strange symbol in lines 128 and 2300 that for some reason my printer rejected), and you can also change your mind over your guess by pressing the delete key.

### **Program notes**

Line 10 : sets colours, into lower case, and selects strange character set.

Line 20 : off to explain the rules.

Line 30 : print heading. The T in The and Thinker is in italics because it is meant to be in lower case. This acts throughout the listing. When you type it in, enter it using the shift key.

Line 40 : top of display, using shifted X char.

Lines 45-80 : print up display of board. Please take very careful note of the REM statements in this part. When REM refers to a character, it means the little chinamen's hats everywhere.

Lines 100-150 : accept only valid characters as guesses, check for delete and left arrow key, play musical (!!) note.

Lines 166-168 : ready to check row?

Lines 169-170 : update number of guesses.

Line 190 : didn't guess it in time!

Line 200 : response to pressing delete key.

Lines 500-605 : check guess, and print up display accordingly.

Lines 800-810 : you guessed right.

Lines 1000-1080 : end of game, display answer, ask for another game.

Lines 2000-2300 : instructions.

Lines 2305-3020 : set up combination for player to guess.

Lines 5000 onwards : various musical pieces.

```
10 POKE 53280,12:POKE 53281,15:POKE 53272,23:POKE
53270,25:PRINT"[CLR,BRN]";
20 GOSUB 2000
30 POKE 53270,8:PRINT"[CLR,GREY] *** THE THINKER
***[BRN]":PRINT
40 PRINTTAB(25);:FORI=1TONC:PRINT" X";:NEXT:PRINT:
REM SHIFTED X FOR GRAPHIC CHAR.
45 PRINTTAB(25);:PRINT"^";:FORI=1TONC*2-1:PRINT"="
;:NEXT:PRINT"^"
```

```

46 REM USE CBM KEY AND A FOR FIRST CHARACTER, CBM
KEY AND S FOR THIRD CHARACTER
50 FORJ=1TO9
60 PRINTTAB(25)"^ ";;FORI=1TONC-1:PRINT " ";;NEXTI
:PRINT"^"
62 REM USE CBM KEY AND Q FOR FIRST CHARACTER, CBM
KEY AND W FOR LAST ONE
65 PRINTTAB(25)"^";;FORI=1TONC*2-1:PRINT"=";;NEXTI
:PRINT"^";NEXTJ
66 REM USE CBM KEY AND Q FOR FIRST CHARACTER, CBM
KEY AND W FOR LAST ONE
70 PRINTTAB(25)"^ ";;FORI=1TONC-1:PRINT " ";;NEXTI
:PRINT"^"
71 REM USE CBM KEY AND Q FOR FIRST CHARACTER, CBM
KEY AND W FOR LAST ONE
75 PRINTTAB(25)"^";;FORI=1TONC*2-1:PRINT"=";;NEXTI
:PRINT"^"
76 REM USE CBM KEY AND Z FOR FIRST CHARACTER, CBM
KEY AND X FOR LAST ONE
80 PRINT"[HOME,2CD,BRN] START GUESSING!"
90 FORP=1TO10
95 G$=""
100 FORI=1TONC
110 GETGU$:IFGU$=""THEN 110
120 FORJ=1TOLEN(C$):IFGU$=MID$(C$,J,1)THEN140
125 IFGU$=CHR$(20)THEN200
128 IF GU$="_"THEN 1000:REM CHECK FOR LEFT ARROW
130 NEXTJ:GOTO 110
140 POKE 1928+I*2-ZZ*80,30:POKE 56200+I*2-ZZ*80,VA
L(GU$)-1
145 G$=G$+GU$
146 GOSUB 5000
150 NEXTI
165 REM END OF ROW
166 GETPP$:IFPP$=CHR$(20)THEN200
167 IFPP$=CHR$(13)THENGU$="14":GOSUB5000:GOTO500:R
EM EVALUATE
168 GOTO 166
169 ZZ=ZZ+1
170 NEXT P
180 REM END OF GUESSES
190 GOTO 1000
200 FORI=1TONC:POKE 56200+I*2-ZZ*80,15:NEXT:GU$="--
3":GOSUB5000:GOTO 95
500 REM CHECK THE GUESS!
505 C1$="":G1$="":X=0:Y=0
510 FORI=1TONC
520 IFMID$(G$,I,1)=MID$(B$,I,1)THENX=X+1:GOTO 530
525 C1$=C1$+MID$(B$,I,1):G1$=G1$+MID$(G$,I,1)
530 NEXTI
540 IFX=NCTHEN800:REM ALL CORRECT!
550 FORI=1TOLEN(C1$)

```

```

555 FORL=1TOLEN(C1$)
560 IFMID$(G1$,I,1)=MID$(C1$,L,1) THENY=Y+1:GOTO 56
6
565 NEXTL
566 C1$=LEFT$(C1$,L-1)+MID$(C1$,L+1)
570 NEXTI
571 IFX=0THEN573
572 FORI=1TOX:POKE 1914+I*2-ZZ*80,122:POKE 56186+I
  *2-ZZ*80,0:NEXTI
573 IFY=0THEN600
574 FORJ=1TOY:POKE 1914+X*2+J*2-ZZ*80,24:POKE56186
  +X*2+J*2-ZZ*80,0:NEXTJ
600 GU$="-12":GOSUB 5000
605 GOTO 169
800 PRINT"[HOME,BRN,CR]YOU GUESSED IT !      ":PRIN
T:PRINT" THE CORRECT ANSWER IS ="
805 GOSUB 6000
810 GOTO 1010
1000 PRINT"[HOME,BRN,CR]SORRY, YOU DIDN'T      ":PRI
NT:PRINT" GET IT. THE ANSWER IS ="
1005 GOSUB 7000
1010 FORI=1TO500:NEXT
1020 FORI=1TONC
1030 POKE 1024+104+I*2,30:POKE 55296+104+I*2,VAL(M
  ID$(B$,I,1))-1
1040 NEXT
1050 PRINT"[CD] PRESS ANY KEY FOR ":PRINT" ANOTHER
  GAME."
1060 GETZ$:IFZ$=""THEN 1060
1070 POKE 53270,25:GOSUB 2200
1075 ZZ=0
1080 GOTO 30
2000 PRINT"WELCOME TO THE THINKER,AN IMPLEMENTATIO
  NOF THE POPULAR BOARD GAME "
2010 PRINT:PRINT"[RVS,RED]MASTERMIND[OFF,BRN] FOR
  THE COMMODORE 64."
2020 PRINT:PRINT"IN THIS GAME, YOU'LL HAVE TO GUES
  S A COMBINATION OF 4,5 OR 6 PIECES.
2030 PRINT:PRINT"THE COMPUTER WILL SET UP A RANDOM
  SET OFCHARACTERS FOR YOU TO GUESS."
2040 PRINT:PRINT"THESE WILL CONSIST OF THE FOLLOWI
  NG SIX COLOURS :=":PRINT
2050 PRINT"[BLK]BLACK [RED]RED [PUR]PURPLE [GRN
  ]GREEN [BLU]BLUE [WHT]WHITE"
2060 PRINT" [BLK]^ [RED]^ [PUR]^ [G
  RN]^ [BLU]^ [WHT]^"
2062 REM THE STRANGE MARKS IN THE LISTING REPRESEN
  T THE UP-ARROW KEY!
2070 PRINT:PRINT"[BRN]YOU CAN CHOOSE HOW MANY PIEC
  ES TO HAVE (4 TO 6), AND WHETHER ";
2080 PRINT"OR NOT COLOURS ARE REPEATED.":GOSUB
  3000

```

```

2100 PRINT"[CLR,BRN]ENTER YOUR GUESS BY PRESSING T
HE APPROP-RIATE KEY ON THE ";
2110 PRINT"TOP ROW OF THE KEYBOARD"
2120 PRINT"FOR EXAMPLE, PRESS '1' IF YOU THINK THE
NEXT SYMBOL IS BLACK, '3' IF ";
2130 PRINT"YOU THINK IT'S RED, AND SO ON."
2140 PRINT:PRINT"JUST PRESS 'INST/DEL' IF YOU WANT
TO CHANGE YOUR MIND."
2145 PRINT:PRINT"PRESS 'RETURN' TO ENTER YOUR GUES
S."
2150 PRINT:PRINT"THE COMPUTER WILL THEN MARK YOUR
GUESS, BY DISPLAYING :="
2160 PRINT:PRINT"RIGHT COLOUR      RIGHT COLOUR
2170 PRINT"RIGHT PLACE      WRONG PLACE  [BLK]X"
2180 POKE 1678,122:POKE55296+654,0
2190 PRINT:PRINT"[BRN]AND NOTHING AT ALL IF IT'S T
HE WRONG COLOUR.":GOSUB 3000
2200 PRINT"[CLR,BRN]WOULD YOU LIKE 4, 5 OR 6 PIECE
S (PRESS 4, 5 OR 6) ?"
2210 GETC$:IFC$=""THEN 2210
2220 IFC$="4"THEN NC=4:GOTO 2255
2230 IFC$="5"THEN NC=5:GOTO 2255
2240 IFC$="6"THEN NC=6:GOTO 2255
2250 GOTO 2210
2255 PRINT"[BLK]";NC;"[BRN]"
2260 PRINT:PRINT"AND WOULD YOU LIKE COLOURS REPEAT
ED OR NOT (PRESS Y OR N) ?"
2270 GET CR$:IFCR$=""THEN 2270
2280 IFCR$="Y"THEN CR=1:PRINT"COLOURS TO BE REPEAT
ED.":GOTO 2300
2290 IFCR$="N"THEN CR=0:PRINT"COLOURS NOT REPEATED
.":GOTO 2300
2295 GOTO 2270
2300 PRINT:PRINT"AND REMEMBER, IF YOU WANT TO GIVE
UP, JUST PRESS [BLK]'_'[BRN]"
2305 C$="135672":B$=""
2310 FORI=1TOUNC
2315 A=INT(RND(.5)*6+1)
2320 IFCRTHEN2340
2325 A$=MID$(C$,A,1)
2330 FORJ=1TOLEN(B$):IFA$=MID$(B$,J,1)THEN2315
2335 NEXTJ
2338 B$=B$+A$:NEXTI:GOSUB3000:RETURN
2340 B$=B$+MID$(C$,A,1):NEXTI:GOSUB3000:RETURN
3000 PRINT:PRINT"[RED]PRESS SPACE TO CONTINUE."
3010 GETSP$:IFSP$<>" "THEN 3010
3020 RETURN
5000 S=54272
5020 POKE S+3,8
5030 POKE S+5,24:POKE S+6,6
5040 POKE S+24,15
5050 POKE S+4,65

```

```

5060 FORK=1T025
5070 POKE S+1,16+VAL(GU$):POKE S,13
5080 NEXTK
5090 FORL=0T024:POKE S+L,0:NEXT
5100 RETURN
6000 S=54272
6010 FORL=0T024:POKE S+L,0:NEXT
6020 POKE S+3,8
6030 POKE S+5,12:POKE S+6,25
6040 POKE S+24,15
6050 POKE S+4,65
6060 FORK=1T03
6070 FORI=1T050:POKE S+1,I:POKE S,I:NEXT
6080 FORI=50T01STEP-1:POKE S+1,I:POKE S,I:NEXT
6085 NEXTK
6090 FORL=0T024:POKE S+L,0:NEXT
6100 RETURN
7000 S=54272
7010 FORL=0T024:POKE S+L,0:NEXT
7020 POKE S+3,8
7030 POKE S+5,32:POKE S+6,72
7040 POKE S+24,15
7050 POKE S+4,65
7060 FORK=1T04
7065 FORI=0T033:POKE S+1,I:POKE S,I:NEXT
7070 FORI=33T00STEP-1:POKE S+1,I:POKE S,I:NEXT
7085 NEXTK
7090 FORL=0T024:POKE S+L,0:NEXT
7100 RETURN

```

## Notes

There are a lot of graphical symbols to watch out for in this game, in particular the left arrow key (line 2300 and line 128), the shifted letters (which have appeared in *italics*), and the setting up of the board in the first part of the program.

However, if you follow all the REMs then you should be all right.

## Machine code routines

This is a collection of little routines, to be entered using the machine code assembler/disassembler given at the back of the book. If you haven't got an assembler, and don't wish either (a) to type it in, or (b) to buy the accompanying cassette, then you'll have to calculate all the memory addresses and all the machine code instructions, and POKE it all in the long way.

The four routines given in this little potpourri will set up a hi-res screen, clear a hi-res screen, jump from a normal screen to a hi-res one, and jump back the other way again. They're all accessed using SYS calls, and in order they are:

SYS 49152,A,B sets up the screen, where A is 0 for an ordinary high resolution screen, or 1 for a multi-colour one, and B is the colour that you're going to get for both the border and the screen background.

SYS 49183 puts you into a high resolution screen.

SYS 49241 clears a high resolution screen, and

SYS 49357 gets you back into a normal screen.

B\*

```

      PC  SR AC XR YR SP
..:70C5 33 00 AD 00 F6
.
C000 20 FD AE      JSR $AEFD
C003 20 EB B7      JSR $B7EB
C006 8A           TXA
C007 8D 20 D0      STA $D020
C00A 8D 21 D0      STA $D021
C00D A5 14         LDA $14
C00F 85 FB         STA $FB
C011 F0 02         BEQ $C015
C013 A2 00         LDX #$00
C015 20 59 C0      JSR $C059
C018 20 80 C0      JSR $C080
C01B 20 A6 C0      JSR $C0A6
C01E A9 3B         LDA #$3B
C020 8D 11 D0      STA $D011
C023 A9 1D         LDA #$1D
C025 8D 18 D0      STA $D018
C028 A5 FB         LDA $FB
C02A F0 05         BEQ $C031
C02C A9 D8         LDA #$D8
C02E 8D 16 D0      STA $D016
C031 A9 80         LDA #$80
C033 85 3B         STA $3B
C035 85 34         STA $34
C037 AD 02 DD      LDA $DD02
C03A 09 03         ORA #$03
C03C 8D 02 DD      STA $DD02
C03F AD 00 DD      LDA $DD00

```

C042	29	FC	AND	#\$FC
C044	09	01	ORA	#\$01
C046	8D	00 DD	STA	DD00
C049	A9	84	LDA	#\$84
C04B	8D	88 02	STA	\$0288
C04E	A9	4F	LDA	#\$4F
C050	8D	11 03	STA	\$0311
C053	A9	C5	LDA	#\$C5
C055	8D	12 03	STA	\$0312
C058	60		RTS	
C059	A0	00	LDY	#\$00
C05B	A9	40	LDA	#\$40
C05D	85	57	STA	\$57
C05F	A9	BF	LDA	#\$BF
C061	85	58	STA	\$58
C063	A9	00	LDA	#\$00
C065	91	57	STA	(\$57),Y
C067	A5	57	LDA	\$57
C069	F0	05	BEQ	\$C070
C06B	C6	57	DEC	\$57
C06D	4C	63 C0	JMP	\$C063
C070	C6	58	DEC	\$58
C072	A5	58	LDA	\$58
C074	C9	9F	CMP	#\$9F
C076	F0	07	BEQ	\$C07F
C078	A9	FF	LDA	#\$FF
C07A	85	57	STA	\$57
C07C	4C	63 C0	JMP	\$C063
C07F	60		RTS	
C080	A0	00	LDY	#\$00
C082	A9	E7	LDA	#\$E7
C084	85	57	STA	\$57
C086	A9	87	LDA	#\$87
C088	85	58	STA	\$58
C08A	8A		TXA	
C08B	91	57	STA	(\$57),Y
C08D	A5	57	LDA	\$57
C08F	F0	05	BEQ	\$C096
C091	C6	57	DEC	\$57
C093	4C	8A C0	JMP	\$C08A
C096	C6	58	DEC	\$58
C098	A5	58	LDA	\$58
C09A	C9	83	CMP	#\$83
C09C	F0	07	BEQ	\$C0A5
C09E	A9	FF	LDA	#\$FF
C0A0	85	57	STA	\$57
C0A2	4C	8A C0	JMP	\$C08A
C0A5	60		RTS	
C0A6	A0	00	LDY	#\$00
C0A8	A9	E7	LDA	#\$E7
C0AA	85	57	STA	\$57
C0AC	A9	DB	LDA	#\$DB



COAE 85 58	STA \$58
COB0 A9 00	LDA #\$00
COB2 91 57	STA (\$57),Y
COB4 A5 57	LDA \$57
COB6 F0 05	BEQ \$COBD
COB8 C6 57	DEC \$57
COBA 4C B0 C0	JMP \$COB0
COBD C6 58	DEC \$58
COBF A5 58	LDA \$58
COC1 C9 D7	CMP #\$D7
COC3 F0 07	BEQ \$C0CC
COC5 A9 FF	LDA #\$FF
COC7 85 57	STA \$57
COC9 4C B0 C0	JMP \$COB0
C0CC 60	RTS
C0CD A9 04	LDA #\$04
C0CF 8D 88 02	STA \$0288
C0D2 AD 02 DD	LDA \$DD02
C0D5 29 FC	AND #\$FC
C0D7 8D 02 DD	STA \$DD02
C0DA A9 1B	LDA #\$1B
C0DC 8D 11 D0	STA \$D011
C0DF A9 C8	LDA #\$C8
COE1 8D 16 D0	STA \$D016
COE4 A9 15	LDA #\$15
COE6 8D 18 D0	STA \$D018
COE9 60	RTS
COEA 00	BRK

It all happens a lot faster than it does in Basic. Hopefully this will whet your appetite for more machine code adventuring, starting with some FILL and PLOT routines perhaps (PLOT is easier!).

We've already seen how to plot a point on the screen using high resolution co-ordinates, albeit in Basic, so since you now know how the code works in Basic, why not have a go at writing such a routine in machine code?



# 7

## General Introduction to Sound

### Musical theory

The Commodore 64 is a virtually unrivalled piece of electronic sophistication when it comes to its musical capabilities. As a synthesiser on a chip it is easily capable of competing with dedicated synthesisers that cost many times more than the Commodore 64. And, of course, the 64 is still a computer.

The chip just referred to is of course the SID chip, technically known as the 6581. We'll be going into more technical details about SID in chapter eight, but we'll start by looking at some traditional musical values.

It is true that you do not have to be a great musician to be able to use the SID chip. Indeed, you hardly have to know anything about music at all, which in some cases (mine) is probably just as well.

### Electronic toy

It can be used purely as an electronic 'toy', although a very powerful one, and it will give as much pleasure to the casual electronic tinkerer as it will to the dedicated computer programmer or to a dedicated user of a synthesiser.

However, since the practical side of using SID is going to be explored in the next two chapters, this introductory chapter will concern itself more with some traditional musical theory (it isn't essential, but it does help, and you will get more out of this chip if you do know the background to it all), and a look at just some of the possibilities Commodore have presented us with.

## **Some musical terms**

Presumably since the dawn of time, people have been making a noise on one kind of musical instrument or another. Those instruments that have survived the passage of time are those that have something special about them, whether in terms of dynamic power, the range of their musical coverage, their sound quality (usually referred to as timbre), or whatever.

Over the years the ever-increasing degree of coaxing which people have given to musical instruments has seen some astonishing advances in both melodic and harmonic structure, since these are largely dependent on the instrument itself, but what has perhaps not advanced so much is the field of rhythmic structure, since that is much more dependent on the skill of the person playing the instrument rather than the instrument itself.






However, with the advent of computers like the Commodore 64 it has become possible to play hitherto unplayable rhythms, and it is possibly this more than any other reason that will eventually see the 64, and others like it, accepted as genuine musical instruments.

## **Traditional music symbols**
















Fortunately for computer buffs and musicians alike, there is a great degree of similarity between the binary system as used by computers and the standard musical notation used to denote the length of time for which a note will be played, or a pause will be held.

Just like computers, music takes its numbers in the form 1,2,4,8,16,32, to denote a whole note, a half note, and so on, all the way down to a thirty-second note, an extremely short note!











Thus whichever way you increase or decrease the tempo of the music, you're always going along in steps of two, like this:

	The <b>WHOLE</b> note	
	The <b>HALF</b> note	(two $\frac{1}{2}$ 's = one whole)
	The <b>QUARTER</b> note	(two $\frac{1}{4}$ 's = one $\frac{1}{2}$ )
	The <b>EIGHTH</b> note	(two $\frac{1}{8}$ 's = one $\frac{1}{4}$ )
	The <b>SIXTEENTH</b> note	(two $\frac{1}{16}$ 's = one $\frac{1}{8}$ )

There are, inevitably, slight complications to this otherwise simple picture, in the form of dotted notes, which are fifty per cent as long again, like this:



	A dotted whole note	= 3		or 2	
	A dotted half note	= 3		or 2	
	A dotted quarter note	= 3		or 2	
	A dotted eighth note	= 3		or 2	
	A dotted sixteenth note	= 3		or 2	



Rests, or pauses, can be regarded in the same light as the notes they are named after, in that they too can be either whole periods of time, or dotted, and the difference between whole rests is always a factor of two. The symbols for rests look like this:



<u>REST</u>		<u>NOTE</u>
	WHOLE	
	HALF	
	QUARTER	
	EIGHTH	
	SIXTEENTH	
<u>Rests.</u>		

Just to finish you off completely and destroy any illusions you might have had that this was going to be all plain sailing, we sometimes have to divide musical notes up into thirds, and these are known as triplets. They are illustrated in the diagram below:

Triplets.

A  $\frac{1}{2}$  note () divided into 3 parts is 

A  $\frac{1}{4}$  note () divided into 3 parts is 

An  $\frac{1}{8}$  note () divided into 3 parts is 

## Movin' in rhythm

Now that we know what all the symbols are, how do we go about using them? We could easily insert a whole lot of random notes into the registers and see what happens, but the resulting cacophony would convince you that there must be something more to music than merely playing sound. There is, of course, and we haven't looked at any items like melody, harmony, or rhythm.

Melody is in the ear of the listener, and what one person finds melodic another may well not. However, that is usually a matter of personal taste, and it is not at all derogatory to say that a piece of music 'has a nice melody'. Some would probably use that as a dismissal, but to most composers it would be praise indeed.

Harmony is just one of the areas that can be explored perhaps more easily on a computer than it can on many other instruments, particularly when you have three independent voices to play with, and three independently selectable waveforms. (Four, if you count noise, but as that is what we're trying to avoid producing we'll stick to the other three.)

### **Movin' in style**

But it is as a rhythmic instrument that, as we said at the beginning, the computer will probably become most widely acceptable. A few simple experiments should serve to convince you that this could be true.

Most people can keep to a fairly reasonable beat. Just try clapping your hands together in a steady rhythm. However, when we move on to, say, a four-one rhythm it gets a little more difficult, but not much, and most people should again be able to keep a fairly steady two-one, three-one or four-one rhythm going.

By four-one, we simply mean that for every fourth beat of, say, the left hand, you also beat the right hand. A two-one would indicate a beat of the right hand for every second beat of the left hand, and so on.

What distinguishes the musicians amongst us is the ability to keep to a totally different rhythm, and then start to sub-divide that rhythm up into several equal-length components.

You try doing a three-two beat! And then dividing that rhythm up into four equal components, and so on. After a short while the hands go totally out of synchronisation and you give up in disbelief that anyone could ever possibly manage to beat in time.

But people do, as any music lover will know. However, there is a limit to what even the most gifted musician can accomplish, and that is why, as we said at the start, the Commodore 64 SID chip can score over many a musical rival.

## **New horizons in sound**

With three independent voices to play with, the 64 is capable, given the right program, of playing something like a 4:7:9 rhythm, an incredibly complex one that we mere mortals would never be able to play properly. But just because it has never been played does not mean that it will not sound pleasing to the ear.

We'll come back to rhythms in the next couple of chapters, but for now, let's take a brief look at the good and bad points of the 6581 SID chip.

## **SID chip overview**

Technically, SID looks something like the diagram opposite.

As you can see from the diagram, SID comes equipped with three voices, each of which has a tone generator to produce the sound, and an envelope generator which affects the structure and hence the volume of that sound. We can also combine some of these tone generators to produce some quite complex sounds.

Most importantly, each voice can be routed through a filter, which is what really sets the 6566 apart from just about any other home computer available. The use of subtle filtering techniques allows some truly amazing sounds to be produced.

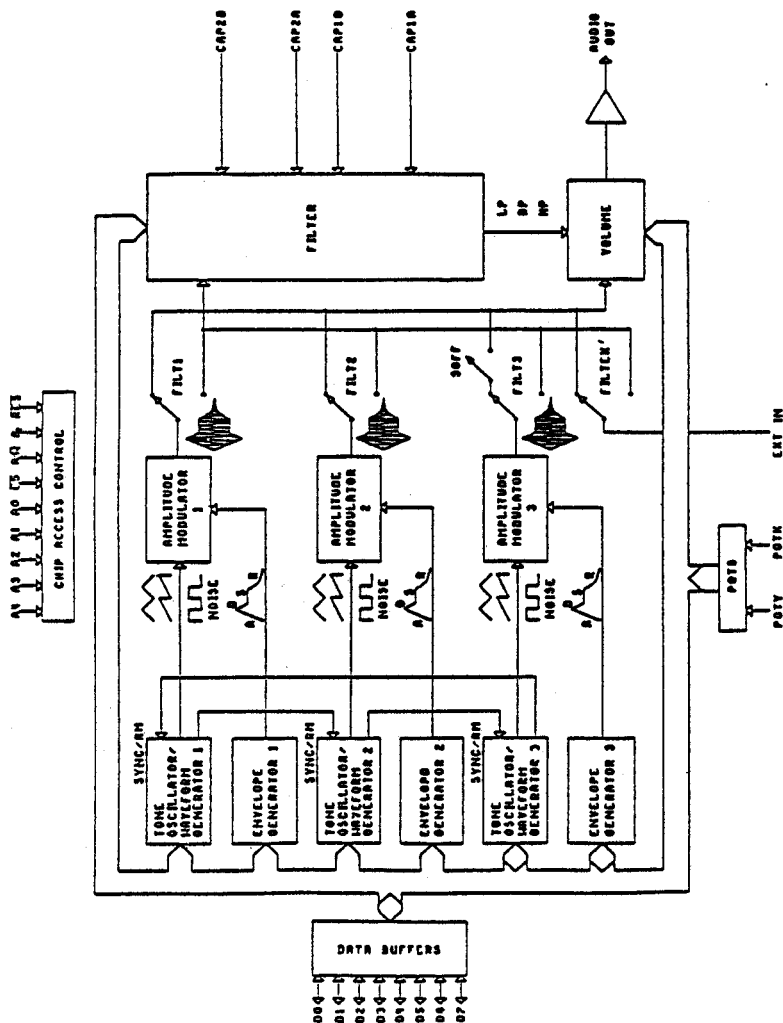
Another thing that all three voices have in common is a master volume control, although this is to be decried rather than applauded. It would have been so much better to have introduced separate volume controls to give SID a truly synthesised polish, even though it would have made programming the chip more difficult.

However, the extra difficulties are as nothing when it comes to trying to hover the volume of one voice at an intermediate level by rapidly switching it on and off.

## **Other features**

There is an external audio output attached to the chip, which allows it to connect up to other SID chips, or link up to some other device like an electric guitar.





Also, you are given a couple of analogue to digital convertors as well, but since these are not electronically wired up to anything else on the chip we won't consider them any further. They could also be used to hitch up some games paddles or something.

## SID registers

As with most of this chapter, the real details will begin to come later,

but it's important that we get the most important terms sorted out first of all, and know precisely what we're talking about.

Each voice has been given 7 registers, which control virtually everything about that voice. In common with all the other registers on the Commodore 64, these are all 8 bit ones, and selecting any combination of bits for any individual register is almost bound to change the sound that you're producing.

There are in addition to this another 8 registers set aside for controlling filtering, master volume, mode selection and so on. Putting them all together in one big map gives a result something like this:

Address	DATA								REG NAME
	D7	D6	D5	D4	D3	D2	D1	D0	
<u>Voice 1</u>									
54272	F <sub>7</sub>	F <sub>6</sub>	F <sub>5</sub>	F <sub>4</sub>	F <sub>3</sub>	F <sub>2</sub>	F <sub>1</sub>	F <sub>0</sub>	FREQ LO
54273	F <sub>15</sub>	F <sub>14</sub>	F <sub>13</sub>	F <sub>12</sub>	F <sub>11</sub>	F <sub>10</sub>	F <sub>9</sub>	F <sub>8</sub>	FREQ HI
54274	PW <sub>7</sub>	PW <sub>6</sub>	PW <sub>5</sub>	PW <sub>4</sub>	PW <sub>3</sub>	PW <sub>2</sub>	PW <sub>1</sub>	PW <sub>0</sub>	PW LO
54275	—	—	—	—	PW <sub>11</sub>	PW <sub>10</sub>	PW <sub>9</sub>	PW <sub>8</sub>	PW HI
54276	NOISE				TEST	RING MOD	SYNC	GATE	CONTROL REG
54277	ATK <sub>3</sub>	ATK <sub>2</sub>	ATK <sub>1</sub>	ATK <sub>0</sub>	DCY <sub>3</sub>	DCY <sub>2</sub>	DCY <sub>1</sub>	DCY <sub>0</sub>	ATTACK/DECAY
54278	STN <sub>3</sub>	STN <sub>2</sub>	STN <sub>1</sub>	STN <sub>0</sub>	RLS <sub>3</sub>	RLS <sub>2</sub>	RLS <sub>1</sub>	RLS <sub>0</sub>	SUSTAIN/RELEASE
<u>Voice 2</u>									
54279	F <sub>7</sub>	F <sub>6</sub>	F <sub>5</sub>	F <sub>4</sub>	F <sub>3</sub>	F <sub>2</sub>	F <sub>1</sub>	F <sub>0</sub>	FREQ LO
54280	F <sub>15</sub>	F <sub>14</sub>	F <sub>13</sub>	F <sub>12</sub>	F <sub>11</sub>	F <sub>10</sub>	F <sub>9</sub>	F <sub>8</sub>	FREQ HI
54281	PW <sub>7</sub>	PW <sub>6</sub>	PW <sub>5</sub>	PW <sub>4</sub>	PW <sub>3</sub>	PW <sub>2</sub>	PW <sub>1</sub>	PW <sub>0</sub>	PW LO
54282	—	—	—	—	PW <sub>11</sub>	PW <sub>10</sub>	PW <sub>9</sub>	PW <sub>8</sub>	PW HI
54283	NOISE				TEST	RING MOD	SYNC	GATE	CONTROL REG
54284	ATK <sub>3</sub>	ATK <sub>2</sub>	ATK <sub>1</sub>	ATK <sub>0</sub>	DCY <sub>3</sub>	DCY <sub>2</sub>	DCY <sub>1</sub>	DCY <sub>0</sub>	ATTACK/DECAY
54285	STN <sub>3</sub>	STN <sub>2</sub>	STN <sub>1</sub>	STN <sub>0</sub>	RLS <sub>3</sub>	RLS <sub>2</sub>	RLS <sub>1</sub>	RLS <sub>0</sub>	SUSTAIN/RELEASE
<u>Voice 3</u>									
54286	F <sub>7</sub>	F <sub>6</sub>	F <sub>5</sub>	F <sub>4</sub>	F <sub>3</sub>	F <sub>2</sub>	F <sub>1</sub>	F <sub>0</sub>	FREQ LO
54287	F <sub>15</sub>	F <sub>14</sub>	F <sub>13</sub>	F <sub>12</sub>	F <sub>11</sub>	F <sub>10</sub>	F <sub>9</sub>	F <sub>8</sub>	FREQ HI
54288	PW <sub>7</sub>	PW <sub>6</sub>	PW <sub>5</sub>	PW <sub>4</sub>	PW <sub>3</sub>	PW <sub>2</sub>	PW <sub>1</sub>	PW <sub>0</sub>	PW LO
54289	—	—	—	—	PW <sub>11</sub>	PW <sub>10</sub>	PW <sub>9</sub>	PW <sub>8</sub>	PW HI
54290	NOISE				TEST	RING MOD	SYNC	GATE	CONTROL REG
54291	ATK <sub>3</sub>	ATK <sub>2</sub>	ATK <sub>1</sub>	ATK <sub>0</sub>	DCY <sub>3</sub>	DCY <sub>2</sub>	DCY <sub>1</sub>	DCY <sub>0</sub>	ATTACK/DECAY
54292	STN <sub>3</sub>	STN <sub>2</sub>	STN <sub>1</sub>	STN <sub>0</sub>	RLS <sub>3</sub>	RLS <sub>2</sub>	RLS <sub>1</sub>	RLS <sub>0</sub>	SUSTAIN/RELEASE
<u>Filter</u>									
54293	—	—	—	—	—	FC <sub>2</sub>	FC <sub>1</sub>	FC <sub>0</sub>	FC LO
54294	FC <sub>10</sub>	FC <sub>9</sub>	FC <sub>8</sub>	FC <sub>7</sub>	FC <sub>6</sub>	FC <sub>5</sub>	FC <sub>4</sub>	FC <sub>3</sub>	FC HI
54295	RES <sub>3</sub>	RES <sub>2</sub>	RES <sub>1</sub>	RES <sub>0</sub>	FILT EX	FILT 3	FILT 2	FILT 1	RES/FILT
54296	3OFF	HP	BP	LP	VOL <sub>3</sub>	VOL <sub>2</sub>	VOL <sub>1</sub>	VOL <sub>0</sub>	MODE/VOL
<u>Misc</u>									
54297	PX <sub>7</sub>	PX <sub>6</sub>	PX <sub>5</sub>	PX <sub>4</sub>	PX <sub>3</sub>	PX <sub>2</sub>	PX <sub>1</sub>	PX <sub>0</sub>	POTX
54298	PY <sub>7</sub>	PY <sub>6</sub>	PY <sub>5</sub>	PY <sub>4</sub>	PY <sub>3</sub>	PY <sub>2</sub>	PY <sub>1</sub>	PY <sub>0</sub>	POTY
54299	O <sub>7</sub>	O <sub>6</sub>	O <sub>5</sub>	O <sub>4</sub>	O <sub>3</sub>	O <sub>2</sub>	O <sub>1</sub>	O <sub>0</sub>	OSCS/RANDOM
54300	E <sub>7</sub>	E <sub>6</sub>	E <sub>5</sub>	E <sub>4</sub>	E <sub>3</sub>	E <sub>2</sub>	E <sub>1</sub>	E <sub>0</sub>	ENV3

You should refer to this diagram, at least in the early days, every time you alter something in one of the registers, so that you can see precisely what you're doing.

We made the point earlier that it is far better to:

**POKE FRED, PEEK (FRED) OR BILL**

than it is to:

**POKE FRED, BILL**

since POKEing FRED with BILL affects not only whatever register(s) go to make up BILL, but also the rest of the ones in memory location FRED as well. The first statement only effects the register(s) in BILL.

As can be seen from the control register diagram, haphazardly altering certain registers can produce somewhat unpredictable results, although there is always the vague possibility that you might stumble across something interesting along the way.

### **Envelope generating**

Each one of the three generators can be set to one of four different waveforms, namely sawtooth, pulse, triangle and noise.

The latter one, noise, has no precise pitch and can come out sounding like anything from the roar of a motor bike to the hissing of a snake, from the sound of waves on the beach to the sound of a cymbal being struck.

Triangle waves tend to give mainly mellow, flute-like sounds, while sawtooth waves have lots of harmonics and are good for impersonating brass or string instruments.

Pulse waves are the richest of the lot, depending on what kind of pulse width you select. With a pulse width of almost zero (or 100 per cent), you get a very thin sound like an oboe, but with a pulse width of around 50 per cent you get a square wave being produced, which sounds very hollow, like a clarinet.

In between these you can produce just about anything, including (provided you have the right filtering set) something remarkably like a human voice.

## **Scales and ranges**

The range of the 6581 chip is usually reported to be eight octaves, although it can go further than that. Notes can be played so low that you won't be able to hear them, even as low as a frequency of about one cycle every 16 seconds. This is not a particularly useful set of notes to play.

The actual frequency that it plays a note at is determined by two registers, termed frequency low and frequency high. Since this gives us a very good control over the frequency of the note that is being played, it allows us to introduce such effects as glissando, by rapidly incrementing or decrementing the frequency of the note so that it appears to glide smoothly from one note to another.

Setting the three voices to be slightly out of tune with each other can also produce some interesting effects, almost choral-like in quality.

## **Ring modulating and synchronising**

Vastly different in name but fairly similar in result, these two techniques allow the various voices to be combined with each other in a number of interesting ways, to produce a mixture of the two tones being played, along with some tones that weren't even there in the first place.

You can only modulate or synchronise one voice with one other voice, but despite that you can get some great effects using these facilities. They make it very easy to simulate such metallic noises as gongs and chimes, as well as producing some very good 'scary science fiction' noises. By varying the frequency of one of the voices while listening to the signal being played, you can hear some pitches rising while others fall.

You can only use ring modulation with the triangular waveform, while synchronisation can be used with any of them, so you'll have to take care if you want to try combining both effects.

## **Filtering**

Mention this to any synthesiser player and he'll begin to realise that perhaps this little machine should be taken seriously after all!

Although filtering doesn't by itself produce a musical note, it does allow you to alter the tone that is being produced by the three voices. Unfortunately you are not allowed to filter voices independently, as there is just the one filter. Still, you can always send two voices to the main output and alter them by changing the pulse width, while at the same time filtering just one of the voices.

There are three different types of filtering that we can use, and these are termed low pass (which reduces in volume everything above a certain cut-off frequency at a rate of 12 decibels per octave, while passing through everything below that frequency as per normal), high pass (which is the reverse, in that everything above a certain frequency gets through as normal, while everything below gets reduced at the 12 decibel rate), and band pass (which passes signals fairly close to a specified frequency and reduces everything else at the usual rate, whether above or below that frequency).

You can also mix the low and high pass filters to produce what is termed a notch filter. This rejects everything near to the specified frequency, but lets through everything that is a certain frequency above or below it: the opposite of band pass.

Finally, the filter also has a resonance control, which determines how strong the effect will be. With a low resonance everything is fairly similar to the effects produced by the tone etc. controls on a stereo (i.e. it all happens gradually), but with a very high resonance the effect is much more severe: like a rock guitarist's wah-wah pedal, which is nothing more than a band pass filter with a very high resonance.

### **Attack/decay/sustain/release**

Together, the four terms listed above combine to form the envelope, or shape, of a musical note, and it is this envelope which determines the timbre of the note being played.

Any or all of these four can be changed for each voice, and in order they do the following:

Attack measures the time it takes for a note to reach maximum volume.

Decay measures the time it takes to decline to an intermediate level.

Sustain holds the note at that level for as long as you want.

Release measures the time it takes for the volume of the note to reach zero i.e. silence.

By varying these, many different musical instruments can be simulated, as we shall see later on.

## **To conclude**

There are other features that we haven't really looked at, such as those attached to voice three that allow you to read the tone generator's output and the envelope generator's output. This could, for instance, be used to produce a tremolo effect by setting voice three to be a triangular wave of very low frequency, reading the waveform value and then adding that to the frequency numbers for voices 1 and/or 2.

The rest is mainly up to you and your willingness to experiment.

However, as yet we don't know what memory locations to experiment with, so without further ado ...!

# 8

## Starting to Play with Sound

### Introduction

From the preceding chapter it can readily be appreciated that we are dealing with a rather special 'dedicated' chip, certainly something much more complex than, say, the Sinclair Spectrum or the Electron sound processors.

Many of the ideas explored in chapter seven will be explained in much more detail in this and the next chapter, and there will be a number of sample programs for you to type in, along with some more ideas for your own experiments with this chip.

As always, however, you've got to walk before you can run, and so some of the next few pages will be spent on giving you all the information you'll need to know in order to start making SID 'talk' to you.

As with the use of graphics on the Commodore 64, it isn't always easy to get started with sound, and some of the topics mentioned here will get only a fairly brief mention. This is not to say that they are all incredibly complex, because that simply isn't true, but they are beyond the scope of anything other than a book dedicated to music and sound on the 64.

However, because the facilities exist they will be mentioned, and at the end of this chapter there will be a round-up of all the features you can find in this chip, what each and every register does, and what most of the bits in these registers do as well.

Armed with that knowledge you can then start exploring the chip for yourself. Filtering voices, synchronising them, modulating them, and generally producing a whole host of effects that will be all the better because you produced them yourself.

Music is, after all, an art not a science, and if everything about music

could be fully explained then there wouldn't be much point in playing it any more!

Nevertheless, all art has got to start somewhere and we've got to start with a bit of science, by giving you the SID chip memory map.

## SID chip memory map

SID allows us access to 28 memory locations, starting at location 54272 and proceeding up to 54300. The table below indicates briefly what each register is capable of controlling.

As usual, we'll adopt our common policy of having a base register, and then detailing everything above that as being the 5th register, 10th register, or whatever. It makes it a lot easier to remember where everything is!

### Register Description (54272 + )

- 00 Low frequency value of note for voice 1
- 01 High frequency value of note for voice 1
- 02 Low pulse rate for voice 1
- 03 High pulse rate for voice 1
- 04 Waveform for voice 1
- 05 Attack/decay for voice 1
- 06 Sustain/release for voice 1
- 07 Low frequency value of note for voice 2
- 08 High frequency value of note for voice 2
- 09 Low pulse rate for voice 2
- 10 High pulse rate for voice 2
- 11 Waveform for voice 2



- 12 Attack/decay for voice 2
- 13 Sustain/release for voice 2
- 14 Low frequency value of note for voice 3
- 15 High frequency value of note for voice 3
- 16 Low pulse rate for voice 3
- 17 High pulse rate for voice 3
- 18 Waveform for voice 3
- 19 Attack/decay for voice 3
- 20 Sustain/release for voice 3
- 21 High frequency cut-off
- 22 Low frequency cut-off
- 23 Turn on filtering
- 24 Set volume for all three voices, plus  
select filter type
- 25 Access to output of envelope generator  
of voice 3
- 26 Access to Y potentiometer reading on pin  
23
- 27 Digitised output from voice 3
- 28 Digitised output from envelope generator  
number 3

So you see, there are a large number of things that we can do with an extremely small number of registers.

But before we can even begin to start playing a note, there are a few more things that we need to know.

If we just concentrate on voice 1 for now, the table above tells us that the volume of that voice (and indeed all the others) is controlled from register 24. So the first couple of lines of a simple program to play a note become:

```
10 V=54272
20 POKE V+24,15
```

Having turned the volume on, we must then select the Attack/decay/sustain/release settings for voice one, as controlled by registers 5 and 6. The previous chapter told us what these settings do, and a little bit about how different settings in each register can affect both the quality and the volume of the note being played. We'll be going into more detail about changing these registers later in the section on envelope generation, but for now it is convenient to think of them as two registers, each of which is split up into two separate nibbles.

In other words, we can think of them as looking like this:

```
Bit      0 1 2 3 4 5 6 7
Value    1 2 4 8 1 2 4 8
```

where the first four bits control the decay (or release) of the note, and the second four control the attack (or sustain) of the note.

If we decide that we want an attack of 4, and a decay of 2, then the value that we must POKE into the appropriate register is found by multiplying the attack value by 16, and adding on the decay value. Hence in this example we'd have to POKE the register with (16\*4 plus 2) or 66.

Different values can be combined, so that if we wanted an attack of 7 and a decay of 5 (thus altering bits 0, 2, 4, 5 and 6), we'd POKE the register with (7\*16 plus 5), or 117.

What do these numbers actually mean? The following tables will show us precisely that.

#### ATTACK/DECAY RATE SETTINGS

	ATTACK/DECAY SETTING	HIGH ATTACK	MEDIUM ATTACK	LOW ATTACK	LOWEST ATTACK	HIGH DECAY	MED. DECAY	LOW DECAY	LOWEST DECAY
VOICE 1	54277	128	64	32	16	8	4	2	1
VOICE 2	54284	128	64	32	16	8	4	2	1
VOICE 3	54291	128	64	32	16	8	4	2	1

### SUSTAIN/RELEASE RATE SETTINGS

SUSTAIN/ CONTROL	RELEASE SETTING	HIGH SUSTAIN	MEDIUM SUSTAIN	LOW SUSTAIN	LOWEST SUSTAIN	HIGH RELEASE	MED. RELEASE	LOW RELEASE	LOWEST RELEASE
VOICE 1	54278	128	64	32	16	8	4	2	1
VOICE 2	54285	128	64	32	16	8	4	2	1
VOICE 3	54292	128	64	32	16	8	4	2	1

We can combine these values in any way we choose, and for the purposes of our program we'll select an attack/decay setting of 68, and a sustain/release setting of 70.

Thus the next two lines of our program become:

```
30 POKE V+5,68
40 POKE V+6,70
```

Slowly but surely we're getting there! The next step is to select our waveform, which is set by altering the content of register 4. This is usually POKEd to be one of the following four values:

17 : selects a triangle waveform

33 : selects a sawtooth waveform

65 : selects a pulse waveform

129 : selects the noise waveform

To play our masterpiece, we'll use the sawtooth waveform, and so the next line of the program becomes:

```
50 POKE V+4,33
```

POKEing that location with one less than the value given above will turn the voice off, without affecting anything else, so that one can rapidly oscillate from on to off.

All we need to know now is what note we are going to play. The two registers that are affected are 0 and 1, and the following table will show us what values to put into those registers for most of the audible notes that SID is capable of producing.

## Table of musical notes

Note	Note-Octave	Hi Freq	Low Freq
0	C-0	1	18
1	C#-0	1	35
2	D-0	1	52
3	D#-0	1	70
4	E-0	1	90
5	F-0	1	110
6	F#-0	1	132
7	G-0	1	155
8	G#-0	1	179
9	A-0	1	205
10	A#-0	1	233
11	B-0	2	6
12	C-1	2	37
13	C#-1	2	69
14	D-1	2	104
15	D#-1	2	140
16	E-1	2	179
17	F-1	2	220
18	F#-1	3	8
19	G-1	3	54
20	G#-1	3	103
21	A-1	3	155
22	A#-1	3	210
23	B-1	4	12
24	C-2	4	73
25	C#-2	4	139
26	D-2	4	208
27	D#-2	5	25
28	E-2	5	103
29	F-2	5	185
30	F#-2	6	16
31	G-2	6	108
32	G#-2	6	206
33	A-2	7	53
34	A#-2	7	163
35	B-2	8	23
36	C-3	8	147
37	C#-3	9	21
38	D-3	9	159
39	D#-3	10	60
40	E-3	10	205
41	F-3	11	114
42	F#-3	12	32
43	G-3	12	216

Note	Note-Octave	Hi Freq	Low Freq
44	G#-3	13	156
45	A-3	14	107
46	A#-3	15	70
47	B-3	16	47
48	C-4	17	37
49	C#-4	18	42
50	D-4	19	63
51	D#-4	20	100
52	E-4	21	154
53	F-4	22	227
54	F#-4	24	63
55	G-4	25	177
56	G#-4	27	56
57	A-4	28	214
58	A#-4	30	141
59	B-4	32	94
60	C-5	34	75
61	C#-5	36	85
62	D-5	38	126
63	D#-5	40	200
64	E-5	43	52
65	F-5	45	198
66	F#-5	48	127
67	G-5	51	97
68	G#-5	54	111
69	A-5	57	172
70	A#-5	61	126
71	B-5	64	188
72	C-6	68	149
73	C#-6	72	169
74	D-6	76	252
75	D#-6	81	161
76	E-6	86	105
77	F6	91	140
78	F#-6	96	254
79	G-6	102	194
80	G#-6	108	223
81	A-6	115	88
82	A#-6	122	52
83	B-6	129	120
84	C-7	137	43
85	C#-7	145	83
86	D-7	153	247
87	D#-7	163	31
88	E-7	172	210
89	F-7	183	25
90	F#-7	193	252
91	G-7	205	133
92	G#-7	217	189
93	A-7	230	176
94	A#-7	244	103

From the above table we can select whatever note is required, and POKE the values into the correct registers. To play note C from the first octave, you can see that the high frequency has a value of 2, and the low frequency a value of 37. So to play the note, we must add the following two lines to our program:

```
60 POKE V+1,2
70 POKE V,37
```

Combining the whole program together, along with a delay line and a closing line to turn everything off again, we get the following: a lot of work for a single note.

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,68
40 POKE V+6,70
50 POKE V+4,33
60 POKE V+1,2
70 POKE V,37
80 FORK=1TO1000:NEXTK
90 FORI=0TO24:POKE V+I,0:NEXT
```

Turning everything off when you've finished is always a good idea, otherwise you'll rapidly be reaching for the headache tablets.

We can use this simple way of building up a note to get the computer to play a little tune. For the time being we'll only use one voice, but even with that voice you'll still be able to get some idea of the effects that the Commodore 64 is generating.

### **Musical tunes**

One way of building up a tune would be to store our note data as data statements, and then read it into a suitable set of variables. Then it would only be a question of POKEing the correct locations with the values stored in the variables, as in the following example.

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,68
40 POKE V+6,70
50 POKE V+4,33
55 READ A,B,C
56 IFA<1THEN95
```

```

60 POKE V+1,B
70 POKE V,C
80 FORI=1TOA*50:NEXTI
85 POKE V+1,0
90 GOTO55
95 FORI=0TO24:POKEV+I,0:NEXT
100 DATA 10,5,185,10,5,185,10,6,108,15,5,105,5,5,1
85,12,6,108
110 DATA 10,7,53,10,7,53,10,7,163,15,7,53,5,6,108,
12,5,185
120 DATA 10,6,108,10,5,185,10,5,105,15,5,185,0,0,0

```

A recognisable little tune.

If you study the listing, you'll see that we're reading in three different variables each time, the second and third being the high and low frequency, and the first just acting as a delay loop, telling us how long to play each note.

From this basic idea, we can begin to play around with the data statements, and the values we're putting into the appropriate registers, to create a host of different sounds.

In this next listing, we're using the pulse waveform, so we've had to put in two lines which set the low and high pulse rates for voice 1. We've also upped everything by a couple of octaves as well.

```

10 V=54272
20 POKE V+24,15
30 POKE V+5,9
40 POKE V+6,0
45 POKE V+2,255
46 POKE V+3,20
50 POKE V+4,65
55 READ A,B,C
56 IFA<1THEN97
60 POKE V+1,B
70 POKE V,C
80 FORI=1TOA*50:NEXTI
95 FORI=0TO23:POKEV+I,0:NEXT
96 GOTO 20
97 FORI=0TO24:POKEV+I,0:NEXT
100 DATA 5,22,227,5,22,227,5,25,177,10,21,154,2,22
,227,7,25,177
110 DATA 5,28,214,5,28,214,5,30,141,10,28,214,2,25
,177,7,22,227
120 DATA 5,25,177,5,22,227,5,21,154,10,22,227,0,0,
0

```

From this, we can do all sorts of things.

The following lines are just some of the ideas that you could incorporate into your own programs, and illustrate how easy it is to alter the sound being produced by the Commodore 64.

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,9
40 POKE V+6,0
45 Z=Z+15
46 POKE V+2,Z:POKE V+3,Z
50 POKE V+4,65
55 READ A,B,C
56 IFA<1THEN97
60 POKE V+1,B
70 POKE V,C
80 FORI=1TOA*50:NEXTI
95 FORI=0TO23:POKEV+I,0:NEXT
96 GOTO 20
97 FORI=0TO24:POKEV+I,0:NEXT
100 DATA 5,22,227,5,22,227,5,25,177,10,21,154,2,22
,227,7,25,177
110 DATA 5,28,214,5,28,214,5,30,141,10,28,214,2,25
,177,7,22,227
120 DATA 5,25,177,5,22,227,5,21,154,10,22,227,0,0,
0
```

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,9
40 POKE V+6,0
50 POKE V+4,17
55 READ A,B,C
56 IFA<1THEN97
60 POKE V+1,B
70 POKE V,C
80 FORI=1TOA*50:NEXTI
95 FORI=0TO23:POKEV+I,0:NEXT
96 GOTO 20
97 FORI=0TO24:POKEV+I,0:NEXT
100 DATA 5,22,227,5,22,227,5,25,177,10,21,154,2,22
,227,7,25,177
110 DATA 5,28,214,5,28,214,5,30,141,10,28,214,2,25
,177,7,22,227
120 DATA 5,25,177,5,22,227,5,21,154,10,22,227,0,0,
0
```



## Musical values

By now you may be wondering how we arrived at all the values listed in the table of musical notes for the high and low frequencies of each note. They are not just numbers plucked out of thin air, refined after experimenting with the SID chip, but are rather based on the physics of sound and the frequency in hertz of each note.

The following table lists the frequencies that can be achieved on the 64 (although the final one listed does go off the scale!).

MUSICAL NOTE	FREQ (Hz)	OSC F# (DECIMAL)	OSC F# (HEX)	MUSICAL NOTE	FREQ (Hz)	OSC F# (DECIMAL)	OSC F# (HEX)
0 C0	16.35	274	0112	48 C4	261.63	4399	1123
1 C#0	17.32	291	0123	49 C#4	277.18	4650	122A
2 D0	18.35	308	0134	50 D4	293.66	4927	133F
3 D#0	19.44	326	0146	51 D#4	311.13	5220	1464
4 E0	20.60	346	015A	52 E4	329.63	5530	159A
5 F0	21.83	366	016E	53 F4	349.23	5859	16E3
6 F#0	23.12	388	0184	54 F#4	370.00	6207	183F
7 G0	24.50	411	019B	55 G4	392.00	6577	19B1
8 G#0	25.96	435	01B3	56 G#4	415.30	6968	1B38
9 A0	27.50	461	01CD	57 A4	440.00	7382	1CD6
10 A#0	29.14	489	01E9	58 A#4	466.16	7821	1ED0
11 B0	30.87	518	0206	59 B4	493.88	8286	205E
12 B#0	32.70	549	0225	60 B#4	523.25	8779	224B
13 C1	34.63	581	0245	61 C#5	554.37	9301	2453
14 C1	36.71	616	0268	62 D5	587.33	9854	267E
15 C#1	38.89	652	029C	63 D#5	622.25	10440	28C8
16 D1	41.20	691	02B3	64 E5	659.26	11060	2B34
17 D#1	43.63	732	02DC	65 F5	698.46	11718	2DC6
18 E1	46.25	776	0308	66 F#5	740.00	12415	307F
19 F1	49.00	822	0336	67 G5	783.99	13153	3361
20 F#1	51.91	871	0367	68 G#5	830.61	13935	366F
21 G1	55.00	923	0399	69 A5	880.00	14764	39AC
22 G#1	58.27	978	03D2	70 A#5	932.33	15642	3D1A
23 A1	61.74	1036	040C	71 B5	987.77	16572	40BC
24 A#1	65.41	1097	0449	72 C6	1046.50	17557	4495
25 B1	69.30	1163	0498	73 C#6	1109.73	18601	48A2
26 B#1	73.42	1232	04D8	74 D6	1174.66	19708	4CFC
27 C2	77.78	1305	0519	75 D#6	1244.51	20897	51BF
28 C2	82.41	1383	0567	76 E6	1318.51	22121	566F
29 C#2	87.31	1465	05B9	77 F6	1396.91	23436	5B8C
30 D2	92.50	1552	0610	78 F#6	1479.98	24830	60FE
31 D#2	98.00	1644	066C	79 G6	1567.98	26306	66CF
32 E2	103.83	1742	06CE	80 G#6	1661.22	27871	6CDD
33 F2	110.00	1845	0735	81 A6	1760.00	29528	735E
34 F#2	116.54	1953	07A3	82 A#6	1864.65	31284	7A34
35 G2	123.47	2071	0817	83 B6	1975.53	33144	8178
36 G#2	130.81	2195	0893	84 C7	2093.00	35115	89E8
37 A2	138.59	2325	0915	85 C#7	2217.46	37203	9153
38 A#2	146.83	2463	099F	86 D7	2349.32	39415	99F7
39 B2	155.56	2610	0A52	87 D#7	2489.01	41759	AB1F
40 B#2	164.81	2763	0ACD	88 E7	2637.02	44242	ACD2
41 C3	174.61	2930	0B72	89 F7	2793.83	46873	BF13
42 C#3	185.00	3104	0C20	90 F#7	2959.95	49660	C1FC
43 D3	196.00	3286	0CD8	91 G7	3135.94	52613	CD8E
44 D#3	207.73	3484	0D9C	92 G#7	3322.44	55741	D9ED
45 E3	220.00	3691	0E5B	93 A7	3520.00	59056	E5B0
46 E#3	232.98	3918	0F46	94 A#7	3729.31	62567	F467
47 F3	246.74	4143	102F	95 B7	3951.06	66286	*1F2F8

This table provides us with a quick and easy way of generating an equal-tempered musical scale. However, since it would take up 192 bytes to store all this in memory, it is not the most memory-effective way of doing things.

It might be better to store all the values for the frequency as one table covering just one octave: this requires a mere 12 bytes to store it. Then, using the fact that notes in different octaves have directly related frequencies (move up one octave and all the frequencies are doubled, move down one and they're all halved), we could readily calculate the frequency of any note in any octave.

This technique is used in one of the programs in chapter nine, where the player of the musical keyboard is allowed to slide up and down from octave to octave, and the computer just calculates whatever the new frequencies should be.

How do we perform these calculations? Let's have a look at some physics.

## **The physics of music**

The sounds that you hear generated by the 64 are nothing more than a series of waves, similar to those of a pond when you throw a stone into it. The ripples showing on the pond are analogous to those produced by generating a sound.

The distance between successive peaks can be accurately measured as a function of time. Thus we determine the interval between successive peaks of the wave passing the same spot. If we call this  $X$  seconds, then the frequency of the wave is denoted as  $(1/X)$ .

In other words, the number of waves passing the same point in one second is called the frequency. This is measured in cycles per second, otherwise known as hertz.

For example, the table above tells us that the pitch for the note middle C is 261.63 hertz. Anything above about 3000 hertz tends to get a little bit painful after a while. Try playing the note A three octaves above middle C (high frequency 230, low frequency 176), and you'll see, or rather hear, what I mean.

To get from a value for the frequency to the values quoted for the high frequency and the low frequency listed earlier, in other words

to get our two values which will determine which note is to be played, we have to do a little bit of mathematics.

If we call the frequency FQ, then the first stage of our equation is:

$$F = \text{INT}(FQ/0.05961)$$

Taking the value F, the high order frequency (call it FH) can be found from the following equation:

$$FH = \text{INT}(F/256)$$

To find the low order frequency (call it FL), we need to repeat the above equation, but don't take the integer part of the number, instead take whatever part comes after the decimal point. So, if the result of dividing F by 256 was something like 5.9856, the part that interests us is the 0.9856.

This is then used as follows:

$$FL = 256 - 256 * 0.9856$$

Using these equations we can then find all the high and low order frequencies, and use them in our programs to produce precisely the notes that we want.

But now, a program!

## **Musical Keyboard**

This program turns the 64 keyboard into a musical one, and allows you to play all three voices.

Any one, or all, of the voices can be altered simply by pressing the return key, which takes you into another part of the program where you alter the waveform of a voice, the shape of the envelope for that voice, and so on. In this way you can experiment with the various sounds that the machine can produce.

The three voices are predefined for you in a set of data statements, but these can easily be changed as you see fit.

It is by no means turning the 64 into a true synthesiser, although it is a step along the way, and we'll come back to this program in the

next chapter and make it do a whole lot more than it does at present.

But since this forms the bare bones around which we'll hang the flesh of that later program, here it is.

### **Program notes**

Line 5 : declare variable V, and turn voice 1 on (all voices are referenced as VO)

Line 20 : go to subroutine to read all the musical data in.

Line 30 : go to subroutine to print on-screen instructions and keyboard.

Line 1000 : check for key press and shift/logo

Line 1010 : if it's all the same as last time then don't do anything.

Line 1020 : get frequency relating to key pressed.

Line 1030 : if no note then turn voices off.

Line 1040 : if numeric key pressed, then 3000.

Line 1045 : if Return pressed, then 4000.

Lines 1050-1060 : if shift or logo pressed, adjust frequency accordingly.

Lines 1070-1080 : calculate frequency.

Lines 1085-1130 : play note for relevant voices.

Lines 2000-2050 : turn voices off.

Lines 3000-3040 : turn on/off relevant voices.

Lines 4000-4024 : on-screen display for altering voices.

Lines 4026-4029 : which voice.

Lines 4030-4036 : to which waveform.

Lines 4037-4250 : rest of alterations.

Lines 5000-5150 : on-screen instructions.

Lines 6000-6120 : data and set-up voices.

```
5 V=54272:VO(0)=1
10 REM FROM AN ORIGINAL IDEA BY RICHARD FRANKLIN
20 GOSUB6000
30 GOSUB5000
1000 K=PEEK(197):PS=PEEK(653)
1010 IFK=LKANDPS=LSTHEN1000
1020 F=N(K):LK=K:LS=PS
1030 IF F=0 THEN 2000
1040 IF (F>0ANDF<9) THEN 3000
1045 IF K=1 THEN 4000
1050 IF PS=1 THEN F=INT(F*2^(1/12))
1060 IF PS=2 THEN F=INT(F/2^(1/12))
1070 F1=INT(F/256)
1080 F2=F-F1*256
1085 FOR I=0 TO 2
1086 IF VO(I)=0 THEN 1125
1090 POKE V+I*7+4,0
1100 POKE V+I*7+4,W(I)*16+RM(I)*4+SY(I)*2+1
1110 POKE V+I*7,F2
1120 POKE V+I*7+1,F1
1125 NEXT I
1130 GOTO 1000
2000 FOR I=0 TO 2
2010 POKE V+I*7,0
2020 POKE V+I*7+1,0
2030 POKE V+I*7+4,W(I)*16
2040 NEXT I
2050 GOTO 1000
3000 F=F-1
3010 FOR I=0 TO 2
3020 VO(I)=(FAND2^I)/2^I
3030 NEXT I
3040 GOTO 1000
4000 POKE 53280,14:POKE 53281,1:PRINT"[PUR]"
4001 PRINT "[CLR]                VOICE 1    VOICE 2    V
OICE 3"
4002 FORI=1TO10:GETKY$:NEXT
4003 PRINT "[CD]WAVEFORM";TAB(12);W$(0);TAB(22);W$(
1);TAB(32);W$(2)
4004 PRINT "[ATT/DEC";TAB(13);AD(0);TAB(23);AD(1);T
AB(32);AD(2)
4006 PRINT "[SUS/REL";TAB(13);SR(0);TAB(23);SR(1);T
AB(32);SR(2)
4008 PRINT "[PULSE #I";TAB(13);PH(0);TAB(23);PH(1);
TAB(32);PH(2)
4010 PRINT "[PULSE L0";TAB(13);PL(0);TAB(23);PL(1);
TAB(32);PL(2)
```

```

4012 PRINT "RING MOD";TAB(13);RM(0);TAB(23);RM(1);
TAB(32);RM(2)
4014 PRINT "SYNC      ";TAB(13);SY(0);TAB(23);SY(1);
TAB(32);SY(2)
4016 PRINT"CDJDO YOU WANT TO CHANGE ANY VALUES (Y
/N)?"
4018 GETCH$;IFCH$="N"THEN30
4020 IFCH$<>"Y"THEN4018
4022 PRINT"CDJWHICH VOICE (1, 2 OR 3)?"
4024 GETVC$;IFVC$=""THEN4024
4026 IFVC$="1"THENPRINT"VOICE 1":VC=0:GOTO 4030
4027 IFVC$="2"THENPRINT"VOICE 2":VC=1:GOTO 4030
4028 IFVC$="3"THENPRINT"VOICE 3":VC=2:GOTO 4030
4029 GOTO 4024
4030 PRINT "CDJWAVEFORM (T, S, P, OR M)?"
4031 GETWF$;IFWF$=""THEN 4031
4032 IFWF$="T"THENPRINT"TRIANGLE":W(VC)=1:W$(VC)="
TRIANGLE":GOTO 4037
4033 IFWF$="S"THENPRINT"SAWTOOTH":W(VC)=2:W$(VC)="
SAWTOOTH":GOTO 4037
4034 IFWF$="P"THENPRINT"PULSE":W(VC)=4:W$(VC)="PUL
SE":GOTO 4037
4035 IFWF$="N"THENPRINT"NOISE":W(VC)=8:W$(VC)="NOI
SE":GOTO 4037
4036 GOTO 4031
4037 INPUT "ATTACK/DECAY";AD(VC):IFAD(VC)<0ORAD(VC
)>255THENPRINT"[2CU]":GOTO 4037
4039 INPUT "SUSTAIN/RELEASE";SR(VC):IFSR(VC)<0ORSR
(VC)>255THENPRINT"[2CU]":GOTO 4039
4041 INPUT "PULSE HI";PH(VC):IFPH(VC)<0ORPH(VC)>25
5THENPRINT"[2CU]":GOTO 4041
4043 INPUT "PULSE LI";PL(VC):IFPL(VC)<0ORPL(VC)>25
5THENPRINT"[2CU]":GOTO 4043
4045 INPUT "RING MOD";RM(VC):IFRM(VC)<0ORRM(VC)>15
THENPRINT"[2CU]":GOTO 4045
4047 INPUT "SYNC";SY(VC):IFSY(VC)<0ORSY(VC)>15THEN
PRINT"[2CU]":GOTO 4047
4049 GOTO 4000
4250 RETURN
5000 POKE 53280,8:POKE 53281,0:POKE 53272,23
5005 PRINT"[YEL,CLR,RVS]***** SING-A-LONG-A-
64 *****[OFF]"
5008 PRINT"[RVS]***** BY SID *****"
5010 PRINT" PLAY USING THE KEYS [RVS]Q W E R T Y U
I"
5020 PRINT"[CD] [RVS]A S D F G
H J K"
5030 PRINT"[CD] [RVS]Z X C V B
N M ,"
5040 PRINT"[CD] TO COVER THREE OCTAVES."
5050 PRINT"[CD] USE THE [RVS]SHIFT[OFF] KEY FOR A

```

```

SHARP,"
5060 PRINT"          [RVS]CBM[OFF]    KEY FOR A FLAT
."
5070 PRINT"[CD]USE THE KEYS [RVS]0 1 2 3 4 5 6 7[O
FF] TO"
5080 PRINT"CHOOSE ANY COMBINATION OF THE THREE"
5090 PRINT"VOICES. THEY ARE SET UP BY USING BINARY
"
5100 PRINT"ARITHMETIC. THEREFORE, VOICE 1 IS TURNE
D";
5110 PRINT"ON USING KEY 1, VOICE 1 AND 3 ARE TURNE
D";
5120 PRINT"ON USING KEY 5,ETC."
5130 PRINT"[CD]USE THE [RVS]RETURN[OFF] KEY TO CHA
NGE THE VALUES"
5140 PRINT"OF THE VOICES.[HOME]"
5150 RETURN
6000 DIM N(64)
6005 FOR I=0 TO 64
6010 READ A
6015 N(I)=A
6020 NEXT I
6025 DATA ,-1,0,0,0,0,0,0
6030 DATA 4,9854,4389,5,2195,4927
6035 DATA 11060,0,6,11718,5530,7,2765,5859
6040 DATA 13153,2463,8,14764,6577,0,3288,7382
6045 DATA 16572,2930,0,17557,8286,1,4143,8779
6050 DATA 0,3691,0,0,0,0,0,0,0,4389,0,0,0
6055 DATA 0,0,0,0,0,2,0,0,3,0
6060 DATA 0,8779,0,0
6070 FOR I=0 TO 2
6080 READ W(I),AD(I),SR(I),PH(I),PL(I),W$(I),RM(I)
,SY(I)
6090 NEXT
6100 DATA 1,102,108,0,0,"TRIANGLE",0,0
6110 DATA 2,96,108,0,0,"SAWTOOTH",0,0
6120 DATA 4,9,0,0,255,"PULSE",0,0
6122 FOR I=0 TO 2
6123 POKE V+7*I+4,W(I)+RM(I)+SY(I)
6124 POKE V+7*I+5,AD(I):POKE V+7*I+6,SR(I):POKE V+
7*I+3,PH(I):POKE V+7*I+2,PL(I)
6126 NEXT
6128 POKE V+24,15
6130 RETURN

```

## Notes

As usual, the up-arrow key causes us problems in a few places, notably lines 1050,1060 and 3020.

The letters in italics are only so because the program has been written in lower case mode, and should be entered as shifted letters when you type the program in.

There are no graphics characters used, so there shouldn't be any great problems in deciphering the rest of the listing.

Of note (sorry!) is the way the frequency values are handled and calculated. They are read in as a table from line 6000 onwards, and are stored in the order of the values returned by PEEK(197) in an array N. Thus if the nth key is pressed, the frequency is found from looking at N(n). The numeric keys cause various voices to be switched on and off, and hence the check in line 1040 to see if one of them has been pressed.

## Using multiple voices

You'll already have seen this kind of thing in action in the last program, but for a few more details, read on.

The kind of principles used in that program, that of playing through a loop and playing all the voices that way, the idea of another loop to turn them all off again, can readily be appreciated, since we're not doing anything too complex with any one voice.

However, playing with ring modulation and synchronisation, as we shall see in the next chapter, can produce some very interesting results, but alas they all require somewhat different techniques in playing the notes.

The following three short examples all use either ring modulation or synchronisation, and illustrate just one way in which these effects can be utilised.

We'll be seeing more of this in the next chapter, but for now three very different sounds. By playing about with the listings, and in particular altering the waveform registers to include or cut out various effects, you'll get a clearer understanding of the ways in which many a strange sound can be produced on the 64.



```

10 V=54272
20 POKE V+24,15
30 POKE V+5,9:POKE V+68,255
40 POKE V+6,0:POKE V+20,70
45 POKE V+2,255
46 POKE V+3,20
50 POKE V+4,67:POKE V+18,33
52 POKE V+14,5:POKE V+15,2
55 READ A,B,C
56 IFA<1THEN97
60 POKE V+1,B
70 POKE V,C
80 FORI=1TOA*50:NEXTI
95 FORI=0TO23:POKE V+I,0:NEXT
96 GOTO 20
97 FORI=0TO24:POKE V+I,0:NEXT
100 DATA 5,22,227,5,22,227,5,25,177,10,21,154,2,22
,227,7,25,177
110 DATA 5,28,214,5,28,214,5,30,141,10,28,214,2,25
,177,7,22,227
120 DATA 5,25,177,5,22,227,5,21,154,10,22,227,0,0,
0

```

```

10 V=54272
20 POKE V+24,15
30 POKE V+5,9:POKE V+19,255
40 POKE V+6,0:POKE V+20,70
50 POKE V+4,23:POKE V+18,33
52 POKE V+14,5:POKE V+15,2
55 READ A,B,C
56 IFA<1THEN97
60 POKE V+1,B
70 POKE V,C
80 FORI=1TOA*50:NEXTI
95 FORI=0TO23:POKE V+I,0:NEXT
96 GOTO 20
97 FORI=0TO24:POKE V+I,0:NEXT
100 DATA 5,22,227,5,22,227,5,25,177,10,21,154,2,22
,227,7,25,177
110 DATA 5,28,214,5,28,214,5,30,141,10,28,214,2,25
,177,7,22,227
120 DATA 5,25,177,5,22,227,5,21,154,10,22,227,0,0,
0

```

```

10 V=54272
20 POKE V+24,15
30 POKE V+5,9:POKE V+19,255:POKE V+12,36
40 POKE V+6,0:POKE V+20,70:POKE V+13,36
46 POKE V+3,A:POKE V+10,15
47 POKE V+2,20:POKE V+9,20

```

```

50 POKE V+4,71:POKE V+18,129:POKE V+11,129
60 FORI=10TO40:POKE V+1,I:POKEV+15,3:POKE V+7,4:NE
XT
70 A=A+10:IFA>250THENA=0
95 FORI=0TO23:POKEV+I,0:NEXT
96 GOTO 30
97 FORI=0TO24:POKEV+I,0:NEXT

```

And now, as a forerunner to the more complex features explored in the next chapter, here is a fairly detailed breakdown of what each of the sound registers in the SID chip actually does. Understanding what the registers do is the key to unlocking the amazing features of the 6581.

## 6581 Register descriptions

We'll now spend a few pages going through each register in detail, using location 54272 as our base register, or register 0, starting with:

### Voice 1 : frequency low/frequency high

These two registers combine together to form a 16 bit number which linearly controls the frequency of voice 1.

This frequency is determined by the following equation:

$$F_{out} = (F_n * F_{clk} / 16777216) \text{ Hz}$$

where  $F_n$  is the 16 bit number in the frequency registers, and  $F_{clk}$  is the system clock applied to the 02 input, pin 6.

Since the Commodore 64 has a one megahertz clock, this formula comes down to:

$$F_{out} = (F_n * 0.05961) \text{ Hz}$$

It should also be noted here that the frequency resolution of the 6581 is such that sweeping from note to note on an even-tempered scale is possible without any noticeable frequency steps.

This allows us to produce such effects as glissando and portamento, where the note sweeps cleanly either up or down the scale, in the case of glissando changing in steps of semitones.

## **Pulse width low and high**

These two registers combine together to form a 12 bit number, which linearly controls the pulse width of the pulse waveform of voice one. Bits 4 to 7 of pulse high are not used, which is why we get just 12 bits and not 16.

This pulse width is determined by the following equation:

$$PW_{out} = (PW_n / 40.95) \%$$

where  $PW_n$  is the 12 bit number in the PW registers.

Again, the pulse width resolution is such that the width can be smoothly swept along without any noticeable stepping effects, as you'll have seen in some of the example programs given earlier.

For constant pulse widths, a value of 0 or 4095 will produce a constant DC output, while a value of 2048 will produce a perfect square wave.

Obviously these features cannot be used without having previously selected the pulse waveform for voice 1. Conversely, setting the pulse waveform for voice 1 and then not setting the pulse width registers won't produce very much in the way of sound either.

## **Control register**

The most important register of them all, containing eight control bits with the following functions:

### **Gate - bit 0**

This controls the envelope generator for voice 1, and when this bit is set to a '1' the envelope generator is triggered and the Attack/Decay/Sustain (or ADS) cycle is begun.

When this bit is reset to a zero, then the release part of the cycle begins.

This envelope generator controls the amplitude of voice 1 as it appears at the audio output, and must therefore be triggered in order for the selected output of voice 1 to be audible.

### **Sync - bit 1**

When set to a '1' this synchronises the fundamental frequency of voice 1 with the fundamental frequency of voice 3, producing what are known as 'hard sync' effects.

Varying the frequency of voice 1 with respect to voice 3 produces a wide range of complex harmonic structures from voice 1 at the frequency of voice 3: this was used in one of the example programs given earlier.

In order for this to take place, obviously voice 3 must be set to some frequency or other, preferably lower than that of voice 1, but naturally higher than zero.

Nothing else connected with voice 3 has any effect on sync.

### **Ring mod - bit 2**

When set to a '1' this bit replaces the triangle waveform of voice 1 with a ring modulation combination of voices 1 and 3: obviously one must previously have selected the waveform of voice 1 to be a triangle one.

Varying the frequency of voice 1 with respect to voice 3 produces a wide range of non-harmonic overtone structures.

Again, nothing else connected with voice 3 has any effect on ring mod.

### **Test - bit 3**

This bit, when set to a '1', resets and holds voice 1 at zero until the bit is cleared.

The noise waveform of voice 1 is also reset, and if a pulse wave has been selected this is held at a DC level.

Normally only used for testing purposes - hence the name - it can be used to synchronise voice 1 to external events.

It can also have a couple of musical applications. Setting this bit to

a '1' instantly clears the voice, whereas using any other method takes at least a couple of milliseconds.

In addition, you can use this to synchronise a number of voices to start at exactly the same time. If, for instance, you were playing a set of rhythms with the three voices, you'd want them all to start at precisely the same moment, otherwise some very non-harmonic effects might occur.

### **Triangle - bit 4**

When this is set to a '1', the triangle waveform is selected for voice 1. This is low in harmonics, and thus produces a mellow, reed-like note.

### **Sawtooth - bit 5**

When this is set to a '1', the sawtooth waveform is selected for voice 1. This is rich in harmonics, and thus produces a brassy, trumpet-like note.

### **Pulse - bit 6**

When this is set to a '1' the pulse waveform is selected for voice 1. The harmonic content of this waveform can be varied by altering the pulse width registers, producing a wide variety of different musical (and not so musical) sounds.

Sweeping through the pulse widths can produce some dynamic effects, and can add a sense of motion to the sound. In the last of the example programs given earlier, breaking into the program and stopping it allows the noise voice to complete its cycle. This sounds remarkably like a train rushing past you.

Rapidly altering from one pulse width to another can also be used to produce some interesting harmonic effects.

### **Noise - bit 7**

When set to a '1', the noise waveform is selected for voice 1. This is a totally random signal which changes at the frequency of voice 1, and thus is of most use in generating purely sound 'effects', like

missiles taking off, engines revving, or vast explosions.

The sound of waves lapping on the beach, or of a cymbal being rapidly hit, can be achieved by sweeping through the different frequencies.

One of the above waveforms must be selected in order for voice 1 to produce any audible sound, although that sound can be turned off without un-selecting a waveform, as the voice at the end is a function of the envelope generator only.

Also, you cannot add more than one waveform together to produce something totally different from the above four.

You are welcome to try, but the most likely result is that voice 1 will be switched off, and can only be reset by the test bit, or by setting pin 5 to low, or '0'.

However, you certainly can combine a waveform with the ring modulation, synchronisation, and other features of this register.

### **Attack/decay**

Bits 4 to 7 of this register, known as ATK0 to ATK3, select an attack rate from 0 to 15 for the voice 1 envelope generator. The attack rate determines how fast the output of voice 1 rises from zero to peak amplitude, when the envelope generator is triggered.

Bits 0 to 3 of this register, known as DCY0 to DCY3, allow you to select a decay rate from 0 to 15 for the envelope generator. The decay cycle comes after the attack cycle, and determines how quickly the output falls from the peak amplitude to some pre-selected intermediate level.

### **Sustain/release**

Bits 4 to 7 of this register, known as STN0 to STN3, allow you to select a sustain level from 0 to 15 for the envelope generator for voice 1. The sustain cycle follows the decay cycle, and determines at what amplitude voice 1 will remain as long as the trigger bit remains set. This is all done on a linear basis, so, for example, a sustain level of 8 would cause voice 1 to sustain at exactly half the peak amplitude reached by the attack cycle.

Bits 0 to 3 of this register, known as RLS0 to RLS3, allow you to select a release rate from 0 to 15 for the envelope generator of voice 1. The release cycle follows the sustain cycle, and determines how rapidly the amplitude of voice 1 will fall from the sustain level to zero amplitude.

The 16 release rates are identical to the decay rates, shown below.

### Envelope rates

The cycling of this envelope generator can be altered at any point in the cycle by the gate bit, as the generator can be gated and released at any time, without restriction.

So if the gate bit is set while half way through an attack cycle, the release cycle will begin immediately, and if the gate is reset again while the release cycle is still continuing, another attack cycle will start from whatever amplitude had been reached during release.

As you might imagine, this gets a bit hairy after a while, but does allow quite complex effects to be achieved.

### Envelope Rates

---

Value		Attack Rate	Decay/Release Rate
Dec	Hex	(Time/Cycle)	(Time/Cycle)
		ms	ms
0	0	2	6
1	1	8	24
2	2	16	48
3	3	24	72
4	4	38	114
5	5	56	168
6	6	68	204
7	7	80	240
8	8	100	300
9	9	250	750
10	A	500	1.5 sec
11	B	800	2.4 sec
12	C	1 sec	3 sec
13	D	3 sec	9 sec
14	E	5 sec	15 sec
15	F	8 sec	24 sec

## **Voices 2 and 3**

### **Voice 2**

The registers \$07 to \$0D control voice 2, and function in the same way as registers \$00 to \$06 for voice 1, with the following two exceptions:

- 1) When SYNC is selected, it synchronises voice 2 with voice 1.
- 2) When RING MOD is selected, it replaces the triangle output of voice 2 with the ring modulated combination of voices 2 and 1.

### **Voice 3**

The registers \$0E to \$14 control voice 3, and function in the same way as registers \$00 to \$06 for voice 1, with the following two exceptions:

- 1) When SYNC is selected, it synchronises voice 3 with voice 2.
- 2) When RING MOD is selected, it replaces the triangle output of voice 3 with the ring modulated combination of voices 3 and 2.

Combining these two effects of modulation and synchronisation can produce some very odd results. You might like to try, for example, synchronising voice 2 with voice 1, and ring-modulating voice 3 with voice 2, or some other weird and wonderful combination of the two.

## **Filtering**

### **Freq Lo/Freq Hi - registers \$15 and \$16**

As bits 3 to 7 of register \$15 are not used, these two combine together to form an 11 bit number which linearly controls the cutoff, or centre frequency of the programmable filter. The approximate cutoff frequency is obtained from the following equation:

$$FC_{out} = ((6.6E-8 + FC_n * 1.28E-8)/C) \text{ Hz}$$

where  $FC_n$  is the 11 bit number in the above two registers and  $C$  is the value of the two filter capacitors connected to pins 1 to 4, or in



our case 2200 picoFarads.

This gives an approximate filter range of 30 Hz to 12 KHz, according to:

$$FC_{out} = (30 + FC_n * 5.8) \text{ Hz}$$

### **Res/Filt - register \$17**

Bits 4 to 7 of this register control the resonance of the filter, where resonance emphasises components of the frequency at the cutoff frequency of the filter, thus causing a sharper sound.

There are 16 resonance settings ranging linearly from no resonance, when this is set to zero, or maximum resonance, when it is set to 15.

As we saw earlier, using a very high resonance is the basis behind a wah-wah pedal, as used by rock guitarists. Pressing the pedal down will raise the cut-off frequency, and releasing the pedal will lower it again. Try it on the 64 and see, by playing a note with very high resonance, and varying the cut-off frequency.

Bits 0 to 3 determine which signals will be routed through the filter.

Bit 0: When this is set to zero, voice 1 appears directly at the audio output, and there is no filtering effect. When set to 1, voice 1 is processed through the filter, and the harmonic content of voice 1 is altered according to the selected filter parameters.

Bit 1: Ditto for voice 2

Bit 2: Ditto for voice 3

Bit 3: Ditto for external audio input on pin 26.

### **Mode/Vol - register \$18**

We've already seen this one as the master volume control, but it actually does a whole lot more.

Bits 0 to 3 are the actual volume settings, and allow you to select an overall volume ranging from 0 (silence) to 15 (maximum).

Bits 4 to 7 select various filter modes and output options.

Bit 4 - When set to a '1', the low pass output of the filter is selected and sent to the audio output. For a given filter input signal, all components of the frequency below the filter cutoff are passed through unaltered, while all those above the cutoff are attenuated at a rate of 12 decibels per octave. This is not as sharp as most dedicated synthesisers (usually 24 decibels per octave), but the effect is still there.

Bit 5 - As above for band pass output, but attenuation above and below the cutoff is at a rate of 6 decibels per octave.

Bit 6 - As above for high pass output, and attenuation below the cutoff is back to 12 decibels per octave.

Bit 7 - When this is set to '1' the output of voice 3 is disconnected from the direct audio path, so setting voice 3 to bypass the filter and setting 3 OFF to a '1' stops voice 3 from ever reaching the audio output. Thus voice 3 can be used for modulation purposes without any extraneous noises coming out and ruining the magnum opus.

These filter modes are additive, in that one can combine a number of different modes at the same time. Playing with, and understanding, these frequency alterations is the key to getting the most out of the 6581.

## 9

# Further Sound Techniques

## Envelope shaping

There are many terms in computing that bear more than passing resemblance to terms in the outside world, and envelope shaping immediately triggers off thoughts of caves hidden deep in a remote valley in Wales, where for years men have gathered and secretly folded envelopes.

However, it's all a bit more mundane on the Commodore 64, and here envelope shaping is taken to mean designing and changing the shape of the musical envelope for the notes that you wish to play.

All good synthesisers give you the ability to change envelope shapes, and SID is no exception. The only difference here between SID and a true synthesiser is that we have to do everything the long way, with a piece of software, rather than just sliding a few dials around on a desk.

Before we talk about changing any envelopes however, it would be a good idea to define a few terms first of all, so that we know precisely what it is that we're altering.

### Musical notes

A musical note goes through four distinct phases as it plays itself out. First there is a rise to the highest volume, and this is termed the attack of the note.

This is sometimes followed by a decay to some intermediate level of volume, and this is indeed referred to as the decay of the note.

The third phase involves keeping at this intermediate volume for a fixed period of time, and this is known as the sustain period of the note.

Finally, the note has to decay away to silence again, and we term this

the release period of the note.

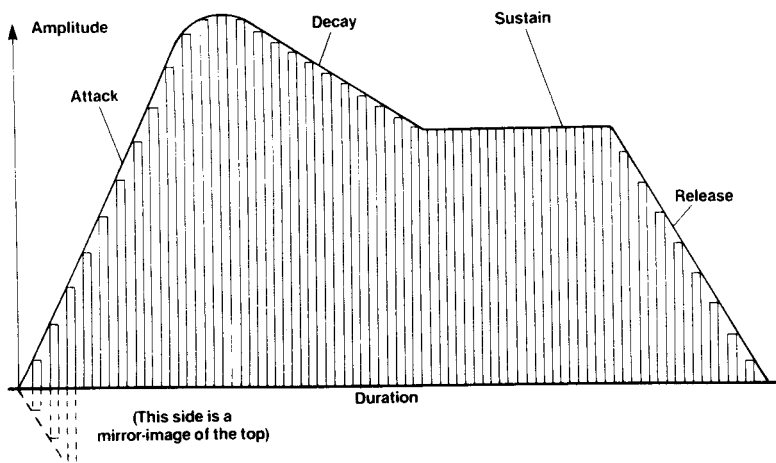
Putting the four together, in a sequence usually referred to as ADSR for the four initial letters of the words involved, gives us the envelope of the note as played by the 64.

This is where synthesisers differ from musical instruments, in that instruments characteristically are unable to alter the envelope of the note that they are playing, at least to any great extent. They don't have to: they are musical instruments precisely because they produce the notes that they do.

Synthesisers, on the other hand, are always having to change envelope shapes of notes as they seek to emulate other sounds, or indeed create new ones.

## Envelope shapes

A typical envelope shape for a musical instrument might look something like this:



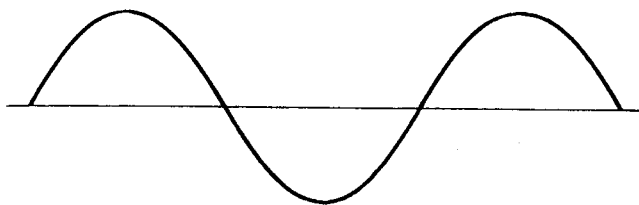
Here we see all four phases of the note looking roughly the same in duration, and by altering one or all four of these characteristics we can dramatically alter the timbre of the note that is being played.

It's good to see on a relatively inexpensive synthesiser like the SID chip that we produce a linear slope for the attack period, but exponential ones for the delay and release. These are built into SID and you cannot alter the way in which they behave. Nevertheless it is good to see that both of these curves are in the chip: most synthesisers use either one or the other, they certainly don't offer you both.

## Waveforms

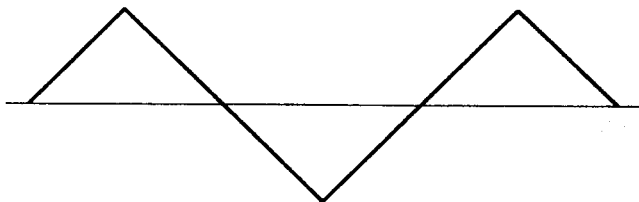
However, we can alter a lot more about a note than merely the shape of its envelope. On the 64 we have control over the waveform of the note as well, and the four waveforms, as we've seen, are termed triangle, sawtooth, pulse and noise.

Graphically, they might look something like this:

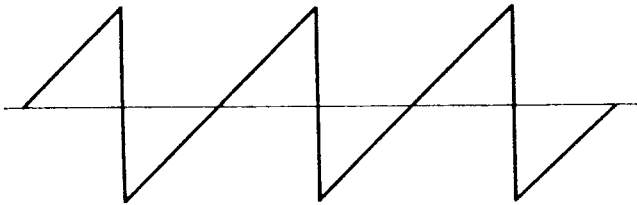


This is a sine wave, and all waveforms can be broken down into a number of different sine waves, all operating at different frequencies.

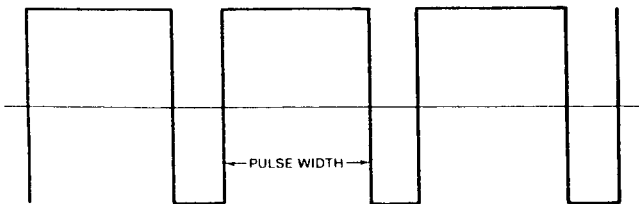
The triangular waveform looks like this:



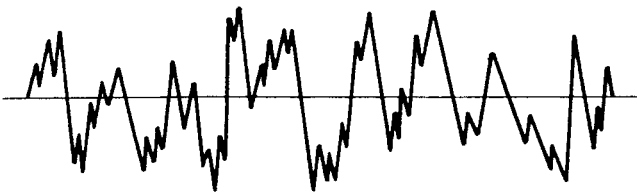
and a sawtooth:



and a pulse wave:



and finally the noise waveform, which is totally random, but might look something like this:



The following short program should serve to demonstrate the principles involved.

## **Attack/decay/sustain/release**

This program is fairly straightforward, but we'll go through it in detail anyway.

It allows you to listen to any single note that the Commodore 64 is capable of playing, before we start filtering, modulating or synchronising other notes.

This is done just by getting you to press a few keys, and see the differences made by everything that the SID chip is capable of altering.

The screen display is nothing complicated. Along the top you'll see the words ATT (for attack), DEC (for decay), SUS (for sustain), REL (for release), FH (which stands for the high frequency), FL (for the low frequency), and finally WF, which indicates the waveform that is currently being used.

By pressing any of the keys 'A', 'D', 'S' or 'R' you will change respectively the attack, decay, sustain or release of the note, and the program will then play the new note.

The change in the value will also be shown on the screen for reference.

Pressing 'W' allows you to step through the four waveforms, and again this change will be shown on the screen and the note played.

To alter the frequency of the note, we have a further four keys to play with. Pressing the up-arrow key will increase the high frequency of the note, pressing the '\*' key will decrease it. Similarly for the low frequency of the note, pressing '+' will increase it by one, and pressing '-' will decrease it by one.

Again, all these changes will be reflected up on the screen.

### **Program notes**

Line 10 : set border, screen and character colours.

Line 20 : declare variable V to be start of video chip.

Line 30 : declare more variables, and GOTO 220.

Line 40 : turn volume to maximum.

Lines 50-60 : select ADSR of note.

Line 70 : check for pulse wave selection.

Lines 80-90 : select frequency of note.

Line 100 : select waveform.

Lines 110-210 : wave for keyboard input and respond accordingly.

Lines 220-230 : print up screen display.

Line 240 : update information on screen.

Line 250 : turn it all off before playing new note.

Line 280 : back to start of loop again.

```
10 POKE 53281,0:POKE 53280,9:PRINT"[CLR,YEL]"
20 V=54272
30 WF=2:A=0:D=9:S=0:R=0:FL=0:FH=12:GOTO 220
40 POKE V+24,15
50 POKE V+5,A*16+D
60 POKE V+6,S*16+R
70 IFWF=4THENPOKEV+3,0:POKEV+2,255
80 POKE V+1,FH
90 POKE V,FL
100 POKE V+4,WF*16+1
110 GETA$:
120 IFA$="+ "THENFL=FL+1:IFFL>255THENFL=255
130 IFA$="^ "THENFH=FH+1:IFFH>255THENFH=255
140 IFA$="A"THENA=A+1:IFA=16THENA=0
150 IFA$="D"THEND=D+1:IFD=16THEND=0
160 IFA$="S"THENS=S+1:IFS=16THENS=0
170 IFA$="R"THENR=R+1:IFR=16THENR=0
180 IFA$="*"THENFH=FH-1:IFFH<0THENFH=0
190 IFA$="W"THENWF=WF*2:IFWF=16THENWF=1
200 IFA$="- "THENFL=FL-1:IFFL<0THENFL=0
210 IFA$=""THEN110
220 PRINT"[HOME]ATT DEC SUS REL FH FL WF"
230 PRINT:PRINT"
"
240 PRINT"[HOME,2CD]"ATAB(4)DTAB(8)STAB(12)RTAB(14)
    FHTAB(18)FLTAB(23)WF*16+1
250 FORI=0TO23:POKE V+I,0:NEXT
280 GOTO40
```



## **Notes**

A straightforward program, but at least it illustrates the concepts involved.

No graphics symbols are used, since this program is primarily concerned with sound rather than graphics, although our usual friend the up-arrow key makes a mess of things in line 120!

## **And more techniques**

If you type that program in and run it you will soon appreciate that there is an awful lot that you can do with your Commodore 64 without ever having to resort to filtering, ring modulation, synchronisation and the like.

The Basic settings for attack, decay, sustain and release can all be altered very easily to produce a wide range of different sounds, and varying the waveforms along with this can also produce some good effects.

However, if we really wish to start turning the 64 into a synthesiser, then we'll have to look a little bit further, so let's start with a look at filtering.

### **Filtering**

We've already stated that there are three main types of filter on the 64, and these are the low pass, high pass and band pass, which let through various frequencies in the tones but dismiss others according to various parameters.

We can combine a high pass and a low pass filter together to get what is termed a notch filter, and with these four factors in mind, we can also alter the resonance of the filter. A high resonance gives a very intense effect, while a low resonance is altogether smoother.

By setting the resonance to be whatever we want, we can then set up various types of filter and, by altering the cutoff frequency so that differing frequencies come through whilst a note is playing, just one note can produce some interesting noises.

And although we haven't got one of the features of some very expensive synthesisers, namely an envelope generator that controls the filter cutoff so that it can automatically rise and fall while the note is playing, we can always get round this with a suitable piece of software.

Finally, each voice can either go through the filter or by pass it completely and go straight to the audio output.

The registers to look for are numbers 21, 22 and 23, as these are the ones that control the cutoff frequency (21 and 22) and the resonance (23).

If bit 0 of register 23 is set to a 1 (i.e. you POKE  $V + 23$ , PEEK( $V + 23$ )OR1) then voice 1 will be routed through the filter. POKEing it with OR2 will set voice 2 through the filter, and POKEing it with OR4 will put voice 3 through the filter.

On a scale of 1 to 15, the resonance is then selected by deciding on what sort of resonance you want (15 being the maximum), and then POKEing the register with your resonance value times 16, plus of course the value for whatever voice, if any, you want to route through.

So, if you wanted maximum resonance and only voice 1 to go through the filter, you would have to POKE  $V + 23$  with 15 times 16 plus 1, or 241.

The type of filter that is used depends on the content of register  $V + 24$ . Normally just set to 15 to control the volume, the top four bits of this register can be altered also to produce one of the four filter types.

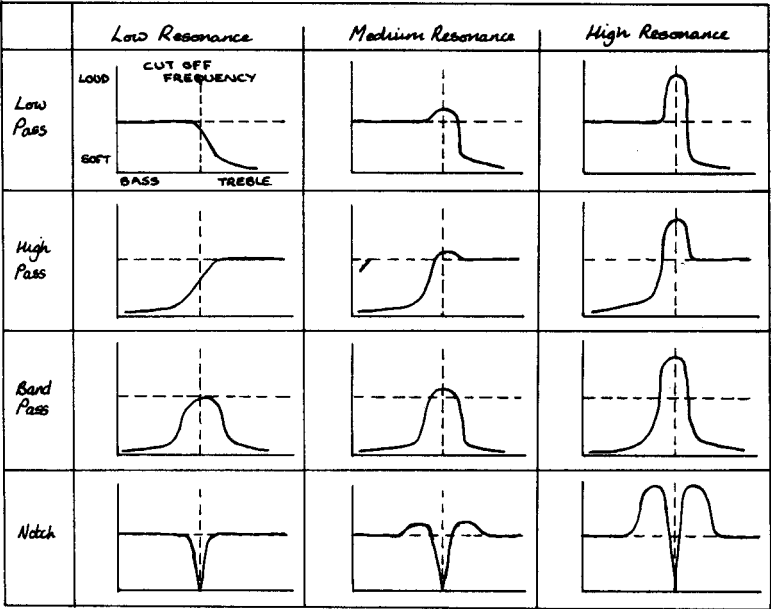
Setting bit 4 (POKE it with 15 plus 16) selects low pass, bit 5 (POKE it with 15 plus 32) band pass, and bit 6 (POKE it with 15 plus 64) high pass. POKEing the register with 15 plus 128, i.e. setting bit 7, stops voice 3 from reaching any audio output, although it is still actively working away.

If we stop it going through the filter as well, then voice 3 never produces any sound at all, but it can still be used for modulating other voices without producing any odd noises of its own in the background.

Finally, to select the cutoff frequency you'll have to refer to the section on that in chapter eight!

To illustrate graphically what's going on, here are a few illustrations

which point out just some of the effects that SID can achieve.



By careful adjustment of any or all of the parameters, the power of SID will slowly begin to become apparent to you. Even if you only produce noise at first, a noise is better than total silence. Well, usually!

### Ring modulation and synchronisation

These two are very closely related, and refer to the way in which different voices interact with each other.

We can set any voice to be modulated with any other voice, although we are limited when it comes to choosing which voice will actually do the modulating.

Similarly we can synchronise any voice with any other voice, but again we are limited in which voice will be doing the synchronising. Still, if you don't suffer from the apparent dislike of voice 3 which some people seem to do for some totally irrational reason (it is actually the most powerful of the three, in that it can control more things that either of the other two), these apparent drawbacks shouldn't present you with any problems.

It is best to let you find out about these two features for yourself, since there isn't much else to say about them really. They just do what they do, and we've already told you which registers to set to get these features going in the previous chapter. So, it's down to the keyboard and a few programs which all produce some very odd noises, but do show what happens when you, for instance, modulate one voice with another one and then synchronise it with a third!

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,9:POKE V+19,255:POKE V+12,36
40 POKE V+6,0:POKE V+20,70:POKE V+13,36
46 POKE V+3,A:POKE V+10,15
47 POKE V+2,20:POKE V+9,20
50 POKE V+4,67:POKE V+18,21:POKE V+11,35
60 POKE V+1,10+A:POKE V+8,15:POKE V+15,20
70 A=A+1:IFA>20THENA=0
80 GOTO 30
97 FORI=0TO24:POKEV+I,0:NEXT
```

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,9:POKE V+19,255:POKE V+12,36
40 POKE V+6,0:POKE V+20,70:POKE V+13,36
46 POKE V+3,A:POKE V+10,15
47 POKE V+2,20:POKE V+9,20
50 POKE V+4,67:POKE V+18,23:POKE V+11,35
60 POKE V+1,10+A:POKE V+8,25-A:POKE V+15,50-A
70 A=A+1:IFA>20THENA=0
80 GOTO 30
97 FORI=0TO24:POKEV+I,0:NEXT
```

```
10 V=54272
20 POKE V+24,15
30 POKE V+5,9:POKE V+19,255:POKE V+12,36
40 POKE V+6,0:POKE V+20,70:POKE V+13,36
46 POKE V+3,20:POKE V+10,15
47 POKE V+2,20:POKE V+9,20
50 POKE V+4,71:POKE V+18,23:POKE V+11,39
60 POKE V+1,10+A:POKE V+8,25-A:POKE V+15,10
70 A=A+1:IFA>20THENA=0
80 GOTO 30
97 FORI=0TO24:POKEV+I,0:NEXT
```

Only by experimenting with all the registers will you get the most out of SID.

## **Pulse widths**

This comes under more or less the same category as synchronisation, since we can tell you what registers to alter to set it in motion, and warn you that not setting pulse widths when choosing the pulse waveform (and vice versa) will cause a few problems, but there's not a lot else to say about it.

It simply modifies the wave pattern of the note being played, and thus causes a different type of note to be produced.

One of the previous example programs did this, by steadily increasing the value stored in the pulse-high register.

It's up to you to find out what values for pulse width suit the type of sounds that you want to make, so again experimentation is the answer.

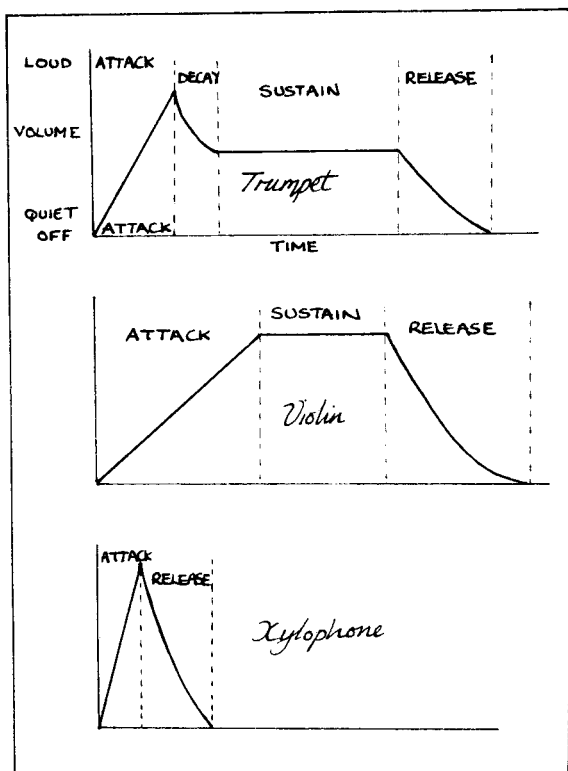
## **Musical instruments**

What, a chapter about synthesised music that dares to mention musical instruments?! Well, since one of the primary functions of a synthesiser is to emulate musical instruments, we might as well try to emulate a few.

Illustrated below are the ADSR diagrams for just a few instruments that you might like to try your hand at, followed by a few suggested settings for the ADSR parameters themselves.

Also important, apart from setting the parameters and getting the envelope right, is the key at which you play the instrument, and the waveform in which it is played. It's no use attempting to play a passable impersonation of a flute with a sawtooth wave: the sound will just not come out right at all!

As usual, experimentation is the key.



### Possible settings for musical instruments

Instrument	Att/Dec	Sus/Rel	Waveform	Pulse
Piano	9	0	Pulse	Lo = 255
Harpsichord	9	0	Sawtooth	—
Trumpet	96	0	Sawtooth	—
Flute	96	0	Triangle	—
Xylophone	9	0	Triangle	—
Organ	9	0	Triangle	—
Accordion	102	240	Triangle	—

And now for a few more programs.

## Advanced Keyboard

This is a variation on the program listed earlier which turned the keyboard into a musical one. This will actually give you some of the power of a synthesiser, although it certainly doesn't go as far as some of the more successful commercially available musical synthesisers (Romik and Quicksilver, for example, have produced good packages for the 64).

Still, it should get you on the right track, and so here goes, with our usual program notes.

### Program notes

Line 5 : declare variable V, and turn voice 1 on (all voices are referenced as VO), as well as setting the octave number.

Line 15 : go to subroutine to read all the musical data in.

Line 20 : go to subroutine to print on-screen instructions and keyboard.

Line 25 : check for key press and shift/logo

Line 28 : if not a special key, then 85.

Line 30 : if DEL key pressed, switch off voice 1.

Line 35 : if CLR key pressed, switch off voice 2.

Line 40 : if pound sign pressed, switch off voice number 3.

Line 45 : if '-' pressed, toggle volume.

Line 50 : volume back to normal again.

Line 55 : if '+' pressed then 420.

Line 60 : if '0' pressed then toggle ring modulation and go to 450.

Line 65 : if '@' pressed, then step down an octave

Line 70 : if up arrow pressed, then step up an octave.

Line 75 : if '\*' pressed then toggle fake glissando mode.

Line 80 : if '=' pressed, then increase fake glissando speed.

Line 85 : if it's all the same as last time then don't do anything.

Line 90 : get frequency relating to key pressed.

Line 95 : if no note then loop back again.

Line 100 : if numeric key pressed, then 225.

Line 105 : adjust frequency if necessary.

Line 110 : check for fake glissando.

Line 115 : if return pressed, then 250.

Lines 120-125 : if shift or logo pressed, adjust frequency accordingly.

Lines 130-135 : calculate frequency.

Lines 140-180 : play note for relevant voices.

Line 185 : remember old frequency.

Lines 190-215 : turn voices off.

Lines 225-240 : turn on/off relevant voices.

Lines 250-300 : on-screen display for altering voices.

Lines 315-340 : which voice.

Lines 345-375 : to which waveform.

Lines 380-415 : rest of alterations.

Lines 420-445 : turn on ring modulation.

Lines 450-500 : fake glissando effect.

Lines 505-590 : on-screen instructions.



Lines 595-715 : data and set-up voices.

```
5 V=54272:VD(0)=1:OC=4
10 REM FROM AN ORIGINAL IDEA BY RICHARD FRANKLIN
15 GOSUB595
20 GOSUB505
25 K=PEEK(197):PS=PEEK(653)
28 IFK<35AND(K=36OR K=37OR K=39OR K=47) THEN85
30 IFK=0THENPOKEV,0:POKEV+1,0:GOTO25
35 IFK=51THENPOKEV+7,0:POKEV+8,0:GOTO25
40 IFK=48THENPOKEV+14,0:POKEV+15,0:GOTO25
45 IFK=43THENR=1-R:POKEV+24,R*15:GOTO25
50 POKE V+24,15
55 IFK=40THEN420
60 IFK=35THENRM=1-RM:GOTO 450
65 IFK=46THENOC=OC/2:IFOC<1THENOC=1:GOTO25
70 IFK=54THENOC=OC*2:IFOC>64THENOC=64:GOTO25
75 IFK=49THENGL=1-GL
80 IFK=53THENGR=GR+1:IFGR>8THENGR=0
85 IFK=LKANDPS=LS THEN25
90 F=N(K):LK=K:LS=PS
95 IF F=0 THEN 25
100 IF (F>0ANDF<9) THEN 225
105 F=F*(4/OC)
110 IFGLANDGR>0ANDZ<>FANDVD(0)=1 THEN455
115 IF K=1 THEN 250
120 IF PS=1 THEN F=INT(F*2^(1/12))
125 IF PS=2 THEN F=INT(F/2^(1/12))
130 F1=INT(F/256)
135 F2=F-F1*256
140 FOR I=0 TO 2
145 IF VD(I)=0THENPOKE V+I*7,0:POKEV+I*7+1,0:GOTO1
80
150 POKE V+I*7+4,0
155 POKE V+I*7+4,W(I)*16+RM(I)*4+SY(I)*2+1
160 IFRM=1THENPOKEV+4,W(0)*16+4
165 POKE V+I*7,F2
170 IFF1>255THEN180
175 POKE V+I*7+1,F1
180 NEXTI
185 Z=F
190 GOTO 25
195 FOR I=0 TO 2
200 POKE V+I*7,0
205 POKE V+I*7+1,0
210 POKE V+I*7+4,W(I)*16
215 NEXT I
220 GOTO 25
225 F=F-1
230 FOR I=0 TO 2
```

```

235 VO(I)=(FAND2^I)/2^I
240 NEXT I
245 GOTO 25
250 POKE 53280,14:POKE 53281,1:PRINT"[PUR]"
255 PRINT "[CLR]          VOICE 1    VOICE 2    VO
ICE 3"
260 FORI=1TO10:GETKY$:NEXT
265 PRINT "[CD]WAVEFORM";TAB(12);W$(0);TAB(22);W$(
1);TAB(32);W$(2)
270 PRINT "ATT/DEC";TAB(13);AD(0);TAB(23);AD(1);TA
B(32);AD(2)
275 PRINT "SUS/REL";TAB(13);SR(0);TAB(23);SR(1);TA
B(32);SR(2)
280 PRINT "PULSE HI";TAB(13);PH(0);TAB(23);PH(1);T
AB(32);PH(2)
285 PRINT "PULSE LO";TAB(13);PL(0);TAB(23);PL(1);T
AB(32);PL(2)
290 PRINT "RING MOD";TAB(13);RM(0);TAB(23);RM(1);T
AB(32);RM(2)
295 PRINT "SYNC      ";TAB(13);SY(0);TAB(23);SY(1);T
AB(32);SY(2)
300 PRINT"[CD]DO YOU WANT TO CHANGE ANY VALUES (Y/
N)?"
305 GETCH$:IFCH$="N"THEN20
310 IFCH$<>"Y"THEN305
315 PRINT"[CD]WHICH VOICE (1, 2 OR 3)?"
320 GETVC$:IFVC$=""THEN320
325 IFVC$="1"THENPRINT"VOICE 1":VC=0:GOTO 345
330 IFVC$="2"THENPRINT"VOICE 2":VC=1:GOTO 345
335 IFVC$="3"THENPRINT"VOICE 3":VC=2:GOTO 345
340 GOTO 320
345 PRINT "[CD]WAVEFORM (T, S, P, OR N)?"
350 GETWF$:IFWF$=""THEN 350
355 IFWF$="T"THENPRINT"TRIANGLE":W(VC)=1:W$(VC)="T
RIANGLE":GOTO 380
360 IFWF$="S"THENPRINT"SAWTOOTH":W(VC)=2:W$(VC)="S
AWTOOTH":GOTO 380
365 IFWF$="P"THENPRINT"PULSE":W(VC)=4:W$(VC)="PULS
E":GOTO 380
370 IFWF$="N"THENPRINT"NOISE":W(VC)=8:W$(VC)="NOIS
E":GOTO 380
375 GOTO 350
380 INPUT "ATTACK/DECAY";AD(VC):IFAD(VC)<0ORAD(VC)
>255THENPRINT"[2CU]":GOTO 380
385 INPUT "SUSTAIN/RELEASE";SR(VC):IFSR(VC)<0ORSR(
VC)>255THENPRINT"[2CU]":GOTO 385
390 INPUT "PULSE HI";PH(VC):IFPH(VC)<0ORPH(VC)>255
THENPRINT"[2CU]":GOTO 390
395 INPUT "PULSE LI";PL(VC):IFPL(VC)<0ORPL(VC)>255
THENPRINT"[2CU]":GOTO 395
400 INPUT "RING MOD";RM(VC):IFRM(VC)<0ORRM(VC)>15T
HENPRINT"[2CU]":GOTO 400

```

```

405 INPUT "SYNC";SY(VC):IFSY(VC)<0ORSY(VC)>15THENP
RINT"[2CU]":GOTO 405
410 GOTO 250
415 RETURN
420 FORI=0TO2
425 IFV(I)=0THEN435
430 POKEV+I*7+4,W(I)*16+2
435 NEXTI
440 IFPEEK(197)=64THEN420
445 GOTO25
450 W(0)=1:POKEV+4,W(0)*16+5:V(2)=1:GOTO25
455 IFZ>FTHENFR=-1:GOTO465
460 FR=1
465 FORI=ZTOFSTEPFR*GR*64
470 F1=INT(I/256)
475 F2=I-F1*256
480 IFRM=1THENPOKEV+4,W(0)*16+5
485 POKE V,F2
490 IFF1>255ORF1<0THEN500
495 POKE V+1,F1
500 NEXTI:Z=I:GOTO130
505 POKE 53280,8:POKE 53281,0:POKE 53272,23
510 PRINT"[VEL,CLR,RVS]***** SING-A-LONG-A-6
4 *****"
515 PRINT"[RVS]***** BY SID *****"
520 PRINT" PLAY USING THE KEYS [RVS]Q W E R T Y U
I"
525 PRINT"[CD] [RVS]A S D F G
H J K"
530 PRINT"[CD] [RVS]Z X C V B
N M ,"
535 PRINT"[CD] TO COVER THREE OCTAVES."
540 PRINT"[CD] USE THE [RVS]SHIFT[OFF] KEY FOR A S
HARP,"
545 PRINT" [RVS]CBM[OFF] KEY FOR A FLAT.
"
550 PRINT"[CD]USE THE KEYS [RVS]0 1 2 3 4 5 6 7[OF
F] TO"
555 PRINT"CHOOSE ANY COMBINATION OF THE THREE"
560 PRINT"VOICES. THEY ARE SET UP BY USING BINARY"
565 PRINT"ARITHMETIC. THEREFORE, VOICE 1 IS TURNED
";
570 PRINT"ON USING KEY 1, VOICE 1 AND 3 ARE TURNED
";
575 PRINT"ON USING KEY 5,ETC."
580 PRINT"[CD]USE THE [RVS]RETURN[OFF] KEY TO CHAN
GE THE VALUES"
585 PRINT"OF THE VOICES.[HOME]"
590 RETURN
595 DIM N(64)
600 FOR I=0 TO 64

```

```

605 READ A
610 N(I)=A
615 NEXT I
620 DATA ,-1,0,0,0,0,0,0
625 DATA 4,9854,4389,5,2195,4927
630 DATA 11060,0,6,11718,5530,7,2765,5859
635 DATA 13153,2463,8,14764,6577,0,3288,7382
640 DATA 16572,2930,0,17557,8286,1,4143,8779
645 DATA 0,3691,0,0,0,0,0,0,0,4389,0,0,0
650 DATA 0,0,0,0,0,2,0,0,3,0
655 DATA 0,8779,0,0
660 FOR I=0 TO 2
665 READ W(I),AD(I),SR(I),PH(I),PL(I),W$(I),RM(I),
SY(I)
670 NEXT
675 DATA 1,102,108,0,0,"TRIANGLE",0,0
680 DATA 2,96,108,0,0,"SAWTOOTH",0,0
685 DATA 4,9,0,0,255,"PULSE",0,0
690 FOR I=0 TO 2
695 POKE V+7*I+4,W(I)+RM(I)+SY(I)
700 POKE V+7*I+5,AD(I):POKE V+7*I+6,SR(I):POKE V+7
*I+3,PH(I):POKE V+7*I+2,PL(I)
705 NEXT
710 POKE V+24,15
715 RETURN

```

## Notes

As usual, the up-arrow key causes us problems in a few places, notably lines 120,125 and 235.

The letters in *italics* are only so because the program has been written in lower case mode, and should be entered as shifted letters when you type the program in.

There are no graphics characters used, so there shouldn't be any great problems in deciphering the rest of the listing.

Of course, it isn't a synthesiser, but it does introduce some of the concepts involved in writing one (which should really be done in machine code), and so that is why it is here.

## Deathtrap

A variation on an old friend, this has been re-written to take into account some of the features we've covered in this book.

The program uses user-defined graphics to display all the characters, and there is also a variety of sound, ranging from rich base notes to explosions, with a motor-bike type sound that has been made by playing about with pulse widths and altering them according to the state of the game.

In the game you control a little shape that wanders about the screen, using the 'A' key to move left, the 'D' key to move right, the 'I' key to move up, and the 'M' key to move down. Pressing any other key will halt the game and give you time to think of a strategy. Just press one of the movement keys to get going again.

As your shape wanders about, he leaves behind a trail on the screen. You must not bump into the trail, as this is the end of your life and the end of the game. To make it more interesting, the computer is also controlling a different shape that is moving about the screen at the same time as you are. This too leaves a trail behind it, and needless to say you can't bump into that either.

The first player who manages to get his opponent to crash into either of the trails wins the game, and a running score is kept throughout.

Obviously the tactic to use is to trap your opponent in a box, but this is frequently easier said than done.

### **Program notes**

Line 5 : set screen, border, and print a little message.

Line 10 : go to routine to set up characters.

Line 15 : and then print the screen instructions.

Lines 20-25 : place the opponents on the screen.

Line 30 : set up sound.

Line 35 : select direction of computer's move, and set you going downwards.

Lines 40-65 : detect change of direction.

Lines 70-100 : moving down.

Lines 105-135 : and moving left.

Lines 140-170 : this time moving to the right.

Lines 175-205 : and finally moving up the screen.

Lines 210-265 : musical introduction.

Line 270 : you lose!

Lines 275-280 : select direction of computer's move.

Line 282 : update noise and play if necessary.

Lines 285-310 : move computer's piece.

Line 315 : ohoh! Can't go that way.

Line 320 : you win!

Lines 325-360 : update noise and play.

Lines 365-395 : computer panicking as it gets boxed in.

Lines 400-440 : explosion as game ends.

Lines 445-475 : update score and next game.

Lines 480-585 : intro and set up game and screen.

Lines 590-660 : define and set up user defined graphics characters.

Line 665 : data for intro tune.

```
5 POKE53280,9:POKE 53281,0:PRINT"[CLR,YEL]JUST HAN
G ON A LITTLE MOMENT, PLEASE...!"
10 GOSUB590
15 GOTO480
20 I=500:POKE 50176+I,128:POKE55296+I,6
25 J=250:POKE 50176+J,129:POKE 55296+J,5
30 GOSUB 325
35 Q$="UDLR":W$=MID$(Q$,RND(.4)*3+1,1):GOTO70
40 A=PEEK(203)
45 IF A=36THEN70
50 IF A=10THEN105
55 IF A=18THEN140
```

```

60 IF A=33THEN175
65 GOTO 40
70 IF I>960THEN I=I-1000
75 FORD=1TODE:NEXT
80 IF PEEK(50176+I+40)<>46 THEN 270
85 I=I+40:POKE50176+I,128:POKE 55296+I,6
90 GOSUB 280
95 A=PEEK(203):IFA=64THEN70
100 GOTO45
105 IF INT(I/40)=I/40THEN I=I+40
110 IF PEEK(50176+I-1)<>46 THEN 270
115 FORD=1TODE:NEXT
120 I=I-1:POKE50176+I,128:POKE 55296+I,6
125 GOSUB 280
130 A=PEEK(203):IFA=64THEN105
135 GOTO45
140 IF INT((I-39)/40)=(I-39)/40THEN I=I-40
145 FORD=1TODE:NEXT
150 IF PEEK(50176+I+1)<>46 THEN 270
155 I=I+1:POKE50176+I,128:POKE55296+I,6
160 GOSUB 280
165 A=PEEK(203):IFA=64THEN140
170 A=PEEK(203):GOTO45
175 IF I<40THEN I=I+1000
180 IF PEEK(50176+I-40)<>46 THEN 270
185 FORD=1TODE:NEXT
190 I=I-40:POKE50176+I,128:POKE55296+I,6
195 GOSUB 280
200 A=PEEK(203):IFA=64THEN175
205 GOTO45
210 V=54272
215 POKE V+14,0:POKE V+4,0:POKEV+5,0:POKEV+6,0
220 POKE V+5,190
225 POKE V+6,0
230 POKE V+24,15
235 READA,B
240 FORI=1T0500:NEXT
245 IFA=0THENRETURN
250 POKE V+4,33
255 POKE V+3,1:POKE V+2,1
260 POKE V+1,A:POKE V,B
265 GOTO235
270 PRINT"[CLR,GRN]TOUGH!":C=C+1:GOTO 400
275 Q$="UDLR":W$=MID$(Q$,RND(.4)*3+1,1):
280 K=INT(RND(.5)*10):IFK>8THEN275
282 XY=XY+1:IFXY/25=INT(XY/25)THENGOSUB352
285 Q=J
290 IFW$="U"THENJ=J-40:IFJ<0THENJ=J+1000
295 IFW$="D"THENJ=J+40:IFJ>1000THENJ=J-1000
300 IF W$="L"THENJ=J-1:IF INT((J+1)/40)=(J+1)/40 T
HEN J=J+40
305 IF W$="R"THENJ=J+1:IF INT((J-40)/40)=(J-40)/40

```

```

    THEN J=J-40
310 IF PEEK(50176+J)=46 THEN POKE50176+J,129:POKE
55296+J,5:S=0:RETURN
315 GOTO 365
320 PRINT"[CLR,GRN]GRRR...":H=H+1:GOTO 400
325 V=54272:Z=0
330 FORL=0TO24:POKE V+L,0:NEXT
335 POKE V+3,SN
340 POKE V+5,40:POKE V+6,146
345 POKE V+24,15
350 POKE V+4,65
352 Z=Z+10
355 POKEV+1,1:POKEV,Z
360 RETURN
365 P$=W$
370 IFF$="U"THENW$="D":J=Q:GOTO285
375 IFF$="D"THENW$="L":J=Q:GOTO285
380 IFF$="L"THENW$="R":J=Q:GOTO285
385 IFF$="R"THENS=S+1
390 IFS=4THENS=0:GOTO320
395 W$="U":J=Q:GOTO285
400 POKE 53280,12:POKE 53281,1
405 V=54272
410 FORL=0TO24:POKE V+L,0:NEXT
415 POKE V+5,17:POKE V+6,130
420 POKE V+24,15
425 POKE V+4,129
430 POKE V+1,3
435 FORP=250TO0STEP-1:POKE V,P:FORPP=1TO5:NEXTPP,P
440 FORL=0TO24:POKE V+L,0:NEXT
445 PRINT"[2CD]SCORE NOW STANDS AT YOU ";H::PRINT:
PRINT"AND THE COMPUTER ";C"
450 FORI=1TO10:GETF$:NEXT
455 PRINT"[CD]ANOTHER GAME (Y OR N)"
460 GET F$:IFF$="" THEN 460
465 IF F$="Y"THEN520
470 IFF$="N" THEN PRINT"[CLR]BYE":FORI=0TO24:POKE
V+I,0:NEXT:END
475 GOTO 460
480 PRINT"[CLR,YEL]WELCOME TO THE GAME OF DEATHTRA
P"
485 GOSUB 210
490 PRINT"[CD]THE OBJECT OF THE GAME IS TO TRAP TH
E "
495 PRINT"COMPUTER SO THAT IT CAN'T MOVE"
500 PRINT"[CD]OF COURSE, IT IS TRYING TO DO THE SA
ME"
505 PRINT"TO YOU!!"
510 PRINT"[CD]PRESS M TO MOVE DOWN. A LEFT, D RIGH
T, "
515 PRINT"AND I UP"
520 PRINT"[CD]DO YOU WANT A FAST, MEDIUM OR SLOW G

```



```

AME"
525 POKE53280.9:POKE 53281,0
530 PRINT"PRESS F, M, OR S"
535 GETD$:IFD$=""THEN 535
540 IF D$="F"THENPRINT"FAST!":DE=0:SN=8:GOTO560
545 IF D$="M"THENPRINT"MEDIUM!":DE=125:SN=10:GOTO5
60
550 IF D$="S"THENPRINT"SLOW!":DE=250:SN=14:GOTO 56
0
555 GOTO 535
560 PRINT"[CD]PRESS SPACE BAR TO START"
565 GETSD$:IFSD$<>" "THEN 565
570 PRINT"[CLR,YEL]";
575 FORI=0TO998:PRINT". ";
580 NEXT:POKE51175,46:POKE 56295,7
585 GOTO 20
590 POKE 56333,127
595 POKE 1,51
600 FORX=0TO1023
605 POKE 53248+X,PEEK(53248+X)
610 NEXT
615 FORX=0TO15
620 READA:POKE54272+X,A
625 NEXT
630 POKE 1,55
635 POKE 56333,129
640 POKE 648,196
645 POKE 56576,4
650 POKE 53272,21
655 DATA 24,90,102,24,24,36,36,36,66,36,189,126,60
,60,66,129
660 RETURN
665 DATA4,208,5,103,4,73,2,6,3,54,0,0

```

## Notes

Amazingly enough, there are no graphics characters used, not even an up-arrow, and since the program is entirely in upper case, we haven't got any strange italicised characters either.

So there should be no problem entering this one.



# 10

## Adding Commands to Basic

### Introduction

In this chapter we'll be looking at some of the ways in which you can enhance the existing Basic in your Commodore 64. Although the method used in this chapter is by no means the only way in which this can be done, it at least has the virtue of working!

There are a number of routines presented here as well, and these could readily be incorporated into one complete package, instead of consisting of a lot of separate little programs as they do now.

Having read this chapter you should be in a position to devise some new commands of your own, and insert these into whatever spare memory space you see fit. Here most of the routines have been written to start at location 49152 (or \$C000), since this is a conveniently empty block of memory on the Commodore 64 at power on.

The commands are presented in the form of machine code disassemblies, in order to give you a better chance of understanding how they work. Basic loaded programs are just so many incomprehensible numbers, whereas these have the machine code mnemonics beside them. Thus a hitherto seemingly random set of numbers turns out to be, in reality, a jump to an internal ROM subroutine, for instance.

If you've got the tape that accompanies this book, or you've typed in the listing for Extramon in the last chapter and got it all working, it will obviously make life easier when it comes to entering the code. However, if your wallet or fingers aren't up to it, then you can still use the commands by converting the numbers as shown in the listings, and hand POKEing them into the machine.

Alternatively, you can convert them into Basic loaders yourself by working out what all those hexadecimal numbers mean, turning them into decimal and storing them as a set of data statements.

Either way, you'll still be able to use them. However, an assembler/disassembler is a powerful tool for the machine code programmer, whether beginning or advanced, and I strongly recommend that one way or another you get that program working.

To aid you in the conversion, here's a handy hex to decimal convertor.

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

These neat little programs will allow you to convert from hex to decimal and back again, and from decimal to binary, although the latter is restricted in size to numbers lying between 0 and 225.

1) Decimal to Hex, where D contains the decimal number, and H\$ = "" before calling this routine. On exit, H\$ contains the hex value :=

```
10 IFDTHENA = INT(D/16):H$ = MID$("0123456789ABCDEF",1 + D-A*16,1) + H$:D = A:GOTO10
```

2) Hex to Decimal, where H\$ contains the hexadecimal value, and D holds the decimal value on exit :=

```
10
D = 0:IFH$ > "" THEN FOR I = 1 TO LEN(H$):A = ASC(MID$(H$,I,1))-48
:D = D*16 + A + (A > 9)*7:NEXT
```

3) Decimal to Binary, where B is the decimal number lying between 0 and 225, and A\$ contains the binary value on exit :=

```
10 A$ = "":FOR I = 0 TO 7:T = B-INT(B/2):2:IFT = 0 THEN A$ = "0" + A$
15 A$ = "1" + A$:B = INT(B/2):NEXTN
```

For now, let's take a look at what Commodore Basic has and hasn't got.

## **Commodore's Basic**

The deficiencies inherent in Commodore Basic are well known. But it's interesting to trace these deficiencies back through time to the very early Commodore machines.

The first Commodore PET, as well as coming complete with its own cassette deck and monitor, and having a paltry 8K of RAM (and also costing some £625 when it first appeared back in 1979!), had what Commodore themselves termed Basic 1.

As a Basic language it was fine at the time, but there were a number of things missing from it. For example, there was no way of accessing the machine code monitor, as it didn't have one built in.

A utility to overcome this soon came on the market, but this took up precious space from the meagre amount of RAM that you had, and so Commodore followers had to wait a couple of years before Basic 2 appeared.

### **Basic 2**

When it did appear it caused instant confusion among the Commodore ranks, since some people were calling it Basic 2, and others referred to it as Basic 3. Seemingly the so-called 'Basic 2' never appeared, and although this particular version of the language was always called Basic 2, theoretically it should have been referred to as Basic 3.

Still, whatever number you gave it it was a great improvement over its predecessor, and did have access to a machine code monitor. Moreover, the ROM installed was now capable of looking after disk drives, something that the earlier machines could not do.

Time went by, Basic 4 appeared, and for a long time it was rumoured that there was to be a fifth version of the language as well, with all the features that existing ones had lacked, of which more in a moment.

However, at the time of writing Basic 5 is in the realms of fantasy, and is not likely to appear now.

## 64 Basic

This brings us to the Commodore 64, sidestepping the Vic along the way.

The version of Basic that they've installed in the machine is that which we referred to earlier as Basic 2/3, although Commodore have apparently now decided that it should be called Basic 2, and indeed this is what the machine greets you with when you turn it on.

However, not only have Commodore taken a retrograde step and installed an old version of their popular Basic language, but they've also managed to take out a great deal of what was already in there.

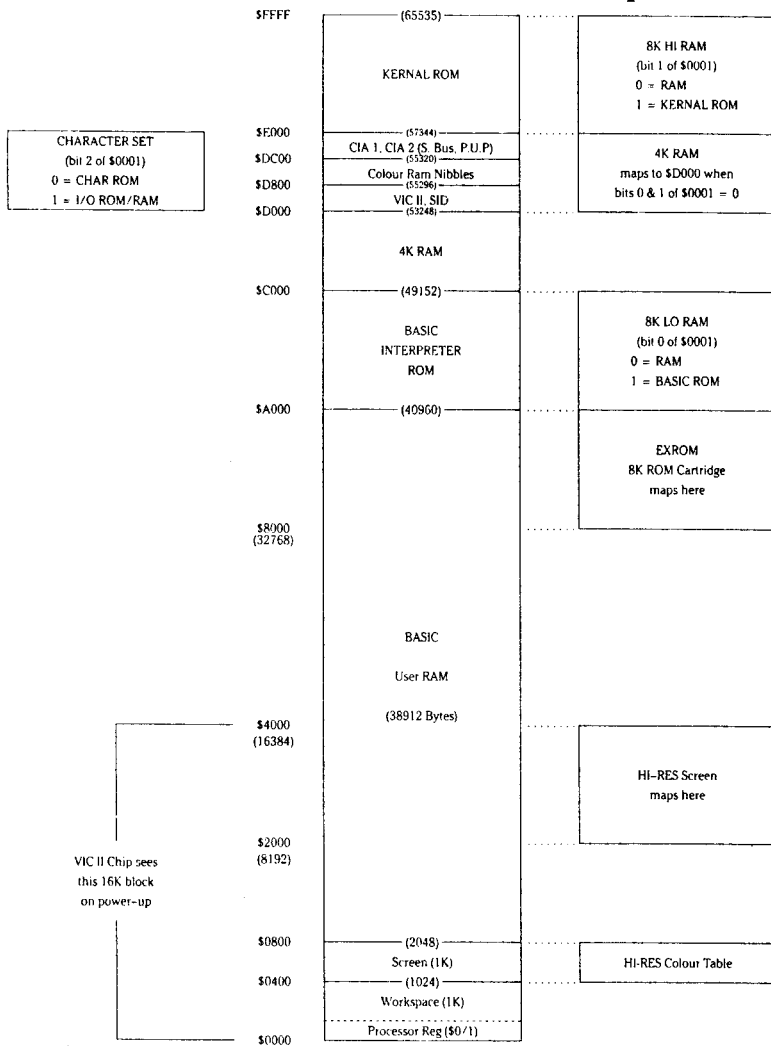
So we see no machine code monitor - and hence the need for programs such as Extramon and the like.

What we are left with instead is an extremely flexible memory management system, but a very poor Basic with which to manage it.

The memory architecture looks something like this:

.

## Commodore-64 Architecture Map



To look after all this requires a lot of work, and to understand it all properly requires even more!

Still, what you buy is what you get, so let's see precisely what we have got.

## **Basic advantages**

The version of Basic in the Commodore 64 is a pretty standard version of what is usually referred to as Microsoft Basic.

This is based on the original Beginners All-purpose Symbolic Instruction Code, from which the language takes its name. This language was devised a number of years ago, and the cracks are now beginning to show, but for a beginner it is still possibly the easiest of languages to learn.

Apart from the interface to machine code, which is not good, the commands you have at your disposal are not too difficult to understand and get to grips with, and owing to the great similarity between Basic words and English words, most beginners can soon start writing programs in Basic.

## **And disadvantages**

However, most beginners also soon come to realise that the version of Basic as supplied by Commodore is sadly lacking in a number of departments.

The concepts of structured programming, the computer flavour of the month, are impossible to simulate on the 64, and there is a distinct lack of such commands as PRINT AT, PRINT USING, and so on.

In particular, when it comes to using graphics and sound, the number of commands is strictly limited to two : PEEK and POKE. No other commands exist to cope with the vast number of PEEKs and POKEs needed to set up a high resolution screen and draw things on it, or to play a few musical notes, or do just about anything with either graphics or sound.

If you want to make music, or display various images on the screen, it has all got to be done the long way, by using a laborious series of POKEs.

Given that this version of Basic is so appalling in these particular departments, it is no wonder that people go to great lengths to try to improve it.



There are now many packages on the market that, in a variety of different ways, have set out to try to improve on the language that we are originally offered.

Whether they succeed in their chosen aims is, of course, a completely different matter, but what they all have in common is that they are adding commands to the existing version of Basic, and through those commands are seeking to make life easier for the person using the machine.

The rest of this chapter will be devoted to showing you one way in which commands could be added, as well as giving you a number of routines to try out for yourself.

But first, the concepts involved.

## **Adding commands: the concepts**

There are many different ways in which you can add commands to Commodore's existing command set. Commands can be added either as words or symbols, or indeed we could also use the function keys: re-define them to be able to accept existing Basic keywords, and then put our new words (or symbols) in their place.

We'll be looking at the two simplest options in this chapter, namely defining various symbols to act as commands, rather than adding new words, and re-defining the function keys to accept these symbols.

These are certainly easier than trying to add new command words to Basic, as this involves altering a lot more things than we are going to do, and for the first time user can seem to be incredibly complicated. So complicated in fact that you probably wouldn't even want to try it!

Still, what we are going to do is fairly straightforward, and shouldn't present any major difficulties.

### **Getting a character**

Anything that you type onto the screen is interpreted and executed by the Commodore 64 as soon as you press the return key. Once this key has been pressed there are a number of routines built into the 64 which will act upon everything that you typed in, and depending on precisely what you typed a number of things will happen.

You can generate a syntax error, and a subroutine exists within the Basic ROM to print out a suitable message and return to await your next input. Since it is in ROM (it starts at location \$AF08) we can't alter it, but there's nothing to stop us copying this ROM into RAM and altering it there, so that SYNTAX ERROR becomes something a lot more meaningful. Or a lot more rude, if you're feeling in that kind of mood!

You could have entered a line of a program, in which case you won't get any error messages (or for that matter any other messages) coming back at all, but a great many pointers inside the machine will have been altered to cope with the new line.

You might have entered a direct command, and in this case the machine will just execute whatever it was that you typed in.

### **Character get routine**

How does the machine know what to do? In other words, how does it interpret what you've typed in? Understanding this is the key to generating our own commands, because if we can intercept the Basic routine that looks after all the commands and alter it, we are then well on the way to adding our own commands into the machine.

The machine knows what to do because of the ROM that's built into it, but there must be a routine somewhere in the machine that looks at what you've typed in and thinks 'ahah!', and then does (or attempts to do) whatever you've told it.

There is indeed such a routine, which lives in locations \$0073 to \$008A (or decimal locations 115 to 138), and this is usually referred to as the CHARGET routine, or character get.

This is the routine that gets a character that you've typed in and acts upon that character.

The routine looks, in its original form, like this:

# CHARACTER GET ROUTINE BEFORE

B\*

```

      PC  SR AC XR YR SP
.:8FEB 33 00 D3 00 F6
.
0073 E6 7A      INC $7A
0075 D0 02      BNE $0079
0077 E6 7B      INC $7B
0079 AD 31 02   LDA $0231
007C C9 3A      CMP #$3A
007E B0 0A      BCS $008A
0080 C9 20      CMP #$20
0082 F0 EF      BEQ $0073
0084 38         SEC
0085 E9 30      SBC #$30
0087 38         SEC
0088 E9 D0      SBC #$D0
008A 60         RTS
.
.
```

What we are going to do is alter that routine so that it no longer behaves in quite the same way.

As it stands at the moment, it interprets everything in the following way:

Locations \$73-\$77 : update the pointer in memory  
locations \$7A and \$7B.

Locations \$79-\$7B : this is the pointer.

Locations \$7C-\$7F : if it's a colon or greater,  
then end.

Locations \$80-\$83 : if it's a space, then loop  
back to start again.

Locations \$84-\$8A : set flags for character type,  
and return from subroutine.

### **Comments**

This routine is the key to adding commands to Basic, since by altering it we can make it jump to some code of our own which will check for a special character, and if that character has been entered then do something! If we find that a special character has not been typed, then it's back to the routine again and carry on as normal.

We'll see later on how we can actually load a program into the computer which, when executed, alters the routine to behave in the way we want.

Instead, a couple of JSRs (jumps to subroutines) will be incorporated in it, and when we've finished with it it will look like this:

AND AFTER!

B\*

PC SR AC XR YR SP  
.;7FC5 33 00 AD 00 F6

.

0073	E6	7A			INC	\$7A
0075	D0	02			BNE	\$0079
0077	E6	7B			INC	\$7B
0079	AD	1E	02		LDA	\$021E
007C	C9	3A			CMP	#\$3A
007E	F0	0A			BEQ	\$008A
0080	C9	20			CMP	#\$20
0082	F0	EF			BEQ	\$0073
0084	20	00	C2		JSR	\$C200
0087	20	00	C1		JSR	\$C100
008A	60				RTS	

.

.

It would be wise, at this point, to make an effort to get Extramon typed up and loaded into the computer, since this will make life a lot easier from now on. Without it we can still proceed with a lot of POKEs, but in order to see precisely what is happening, Extramon is a great help.

## Altering the CHARGET routine

When everything is running normally, on pressing the Return key the system will come out of ROM into this routine to fetch the next character of Basic text, then trundle back into ROM again to ponder on its next move.

What will happen now is that the system will come out of ROM, to our changed subroutine, and when it hits the first JSR command it will jump to the routine sitting at location \$C200 onwards. This will determine whether or not we're going to be interpreting a special command, and if we are jump somewhere else to process it.

If we're not, then the system goes back to the altered CHARGET routine, and finds that it now has to make yet another jump, this time to location \$C100. This is simply a direct copy of what used to exist in the portion of CHARGET that we have changed, so that execution can continue as normal in the event of a special character not being found.

After that, it returns into ROM again to work out what will happen next.

The program to alter the CHARGET routine sits at locations \$C10B onwards, and together with the direct replacement for the altered parts, which starts at location \$C100, it all looks like this :

```
B*
      PC  SR  AC  XR  YR  SP
.:8FEB 33 00 D3 00 F6
.
C100 C9 3A      CMP  ##3A
C102 B0 06      BCS  $C10A
C104 38         SEC
C105 E9 30      SBC  ##30
C107 38         SEC
C108 E9 D0      SBC  ##D0
C10A 60         RTS
C10B A9 20      LDA  ##20
C10D B5 B4      STA  $B4
```

C10F	85	87	STA	\$87
C111	A9	00	LDA	#\$00
C113	85	85	STA	\$85
C115	85	88	STA	\$88
C117	A9	C2	LDA	##C2
C119	85	86	STA	\$86
C11B	A9	C1	LDA	##C1
C11D	85	89	STA	\$89
C11F	A9	F0	LDA	##F0
C121	85	7E	STA	\$7E
C123	A9	04	LDA	#\$04
C125	85	7A	STA	\$7A
C127	85	7B	STA	\$7B

.  
.

The next routine that we need is the one to separate the extra code and the processing of that code from ordinary Basic. In this routine we check the current character being processed against a table stored at locations \$C300 onwards, and if we find what we're looking for, branch to the appropriate subroutine by reading the most significant byte and least significant byte of the subroutine address from a table which is stored immediately after the character data.

B\*  
PC SR AC XR YR SP  
.;371A 33 00 02 00 F6  
.  
C200 08 PHP  
C201 86 04 STX \$04  
C203 A2 04 LDX ##\$04  
C205 DD 00 C3 CMP \$C300,X  
C208 F0 07 BEQ \$C211  
C20A CA DEX  
C20B 10 FB BPL \$C205  
C20D A6 04 LDX \$04  
C20F 28 PLP  
C210 60 RTS  
C211 BD 06 C3 LDA \$C306,X  
C214 BD 1E C2 STA \$C21E  
C217 BD 08 C3 LDA \$C308,X  
C21A BD 1F C2 STA \$C21F  
C21D 20 00 C0 JSR \$C000  
C220 20 74 A4 JSR \$A474  
.  
.

To explain what's happening, the program first of all saves the current status register onto the stack, and the current value held in the X register into location \$0004 in the event of not finding a special character.

If that is the case, then everything is read back into the appropriate registers and it's off to the CHARGET routine again.

### Finding special characters

However, if a special character is found then we branch out to location \$C211 where we get the least significant byte and the most significant byte from our table. These are then stored at the appropriate registers, and then the program branches off to the subroutine to carry out the command.

The characters, and their LSBs and MSBs look like this :

```

B*
      PC  SR  AC  XR  YR  SP
.;26F4 33 00 DC 00 F6
.
C300 5F                ???
C301 21 21            AND ($21,X)
C303 21 21            AND ($21,X)
C305 00                BRK
C306 00                BRK
C307 C0 C0            CPY ##C0
.
.
```

This may not look very sensible as a disassembly, but it's the data that we're after, not the annotations.

The only thing we need now is a routine to execute, and in this case we've used an OLD routine. This can be used without the rest of this code by just typing in SYS49152, and it will then recover any program lost after a NEW command had been issued.

On the other hand, there's something infinitely more satisfying about seeing your own code being executed at the press of a key, rather than typing in boring old SYS commands all the time.



B\*

```
PC SR AC XR YR SP
.;6F9F 33 00 87 00 F6
.
C000 A5 2B      LDA $2B
C002 A4 2C      LDY $2C
C004 85 22      STA $22
C006 84 23      STY $23
C008 A0 03      LDY #$03
C00A C8         INY
C00B B1 22      LDA ($22),Y
C00D D0 FB      BNE $C00A
C00F C8         INY
C010 98         TYA
C011 18         CLC
C012 65 22      ADC $22
C014 A0 00      LDY #$00
C016 91 2B      STA ($2B),Y
C018 A5 23      LDA $23
C01A 69 00      ADC #$00
C01C C8         INY
C01D 91 2B      STA ($2B),Y
C01F 88         DEY
C020 A2 03      LDX #$03
C022 E6 22      INC $22
C024 D0 02      BNE $C02B
C026 E6 23      INC $23
C028 B1 22      LDA ($22),Y
C02A D0 F4      BNE $C020
C02C CA         DEX
C02D D0 F3      BNE $C022
C02F A5 22      LDA $22
C031 69 02      ADC #$02
C033 85 2D      STA $2D
C035 A5 23      LDA $23
C037 69 00      ADC #$00
C039 85 2E      STA $2E
C03B 60         RTS
.
.
```

Now that we've got everything together, it only remains to run the program by going to the various parts of it, and then, just by pressing the left arrow key and Return, we can instantly recover any program that may have been lost due to an accidental NEW.

To add yet more commands, you'll need to store more data for characters, and more data for LSBs and MSBs at locations \$C3000 and onwards (or anywhere else for that matter, as long as the program

is pointed to the correct location!). The data for the characters is just the ASCII code for that character.

If, however, your forte is typing words rather than symbols, the following program shows how you might use the word BAK to get back a program, rather than typing in the left-arrow key.

```

B*
    PC SR AC XR YR SP
.;8D55 31 72 9F 00 F6
.
C200 0B          PHP
C201 86 04          STX $04
C203 A2 00          LDX #$00
C205 DD 00 C3      CMP $C300,X
C208 F0 26          BEQ $C230
C20A CA            DEX
C20B 10 FB          BPL $C205
C20D A6 04          LDX $04
C20F 28            PLP
C210 60            RTS
C211 BD 06 C3      LDA $C306,X
C214 8D 1E C2      STA $C21E
C217 BD 08 C3      LDA $C308,X
C21A 8D 1F C2      STA $C21F
C21D 20 C0 FF      JSR $FFC0
C220 E6 C9          INC $C9
C222 D0 02          BNE $C226
C224 E6 CA          INC $CA
C226 28            PLP
C227 A2 00          LDX #$00
C229 A1 C9          LDA ($C9,X)
C22B A6 04          LDX $04
C22D 60            RTS
C22E 00            BRK
C22F 00            BRK
C230 E6 7A          INC $7A
C232 D0 02          BNE $C236
C234 E6 7B          INC $7B
C236 A2 00          LDX #$00
C238 A1 7A          LDA ($7A,X)
C23A 38            SEC
C23B E9 41          SBC #$41
C23D F0 03          BEQ $C242
C23F 4C 5A C2      JMP $C25A
C242 E6 7A          INC $7A
C244 D0 02          BNE $C248
C246 E6 7B          INC $7B
C248 A2 00          LDX #$00
C24A A1 7A          LDA ($7A,X)

```

C24C	38		SEC
C24D	E9	4B	SBC #\$4B
C24F	F0	03	BEQ \$C254
C251	4C	0B AF	JMP \$AF0B
C254	20	00 C0	JSR \$C000
C257	20	74 A4	JSR \$A474

## Early experiments

You could start your experiments with adding commands by trying to interface the graphics routines given earlier to act at the press of a key. Where commands will need parameters added to them, you could have another table of addresses to interpret things like !1, !2, !3 and so on where, having found that an '!' symbol has been entered, you then check to see what the next number is and go to the correct subroutine.

Where you'll need other parameters to be separated by commas, it is most practical to go to the internal ROM routines and let them do the checking, as was done with the graphics routine for setting up a high resolution screen in a suitable colour.

### Function keys

We said earlier that we'd be giving you a program to use the function keys, and here it is. As it stands, it allows you to define the function keys to be any of the existing Basic keywords, although of course if you add your own commands you can also define a key to be one or more of those as well.

When you run the program by typing in SYS 49152, the prompt F1? will appear, at which point you enter whatever you want function key 1 to be (e.g. PRINT). If you want to make it equivalent to typing in PRINT and then hitting the Return key, enter PRINT followed by the left arrow key, which has been used in this program to stand for the Return key.

B\*

PC SR AC XR YR SP  
.:B7DB 33 00 C0 00 F4

C000	A9	00		LDA	#\$00
C002	AA			TAX	
C003	9D	00	C2	STA	\$C200,X
C006	9D	00	C3	STA	\$C300,X
C009	9D	00	C4	STA	\$C400,X
C00C	E8			INX	
C00D	D0	F4		BNE	\$C003
C00F	85	FB		STA	\$FB
C011	A9	C2		LDA	#\$C2
C013	85	FC		STA	\$FC
C015	A9	31		LDA	#\$31
C017	85	FD		STA	\$FD
C019	A9	85		LDA	#\$85
C01B	85	FE		STA	\$FE
C01D	A9	0D		LDA	#\$0D
C01F	20	D2	FF	JSR	\$FFD2
C022	A9	46		LDA	#\$46
C024	20	D2	FF	JSR	\$FFD2
C027	A5	FD		LDA	\$FD
C029	20	D2	FF	JSR	\$FFD2
C02C	A9	3D		LDA	#\$3D
C02E	20	D2	FF	JSR	\$FFD2
C031	A9	3F		LDA	#\$3F
C033	20	D2	FF	JSR	\$FFD2
C036	20	CF	FF	JSR	\$FFCF
C039	48			PHA	
C03A	A0	00		LDY	#\$00
C03C	A5	FE		LDA	\$FE
C03E	91	FB		STA	(\$FB),Y
C040	68			PLA	
C041	20	85	C0	JSR	\$C085
C044	C9	0D		CMP	#\$0D
C046	F0	11		BEQ	\$C059
C048	C9	5F		CMP	#\$5F
C04A	D0	02		BNE	\$C04E
C04C	A9	0D		LDA	#\$0D
C04E	91	FB		STA	(\$FB),Y
C050	20	85	C0	JSR	\$C085
C053	20	CF	FF	JSR	\$FFCF
C056	4C	44	C0	JMP	\$C044
C059	E6	FD		INC	\$FD
C05B	A5	FD		LDA	\$FD
C05D	29	01		AND	#\$01
C05F	D0	0A		BNE	\$C06B
C061	18			CLC	
C062	A5	FE		LDA	\$FE
C064	69	04		ADC	#\$04
C066	85	FE		STA	\$FE

C068	4C	72	C0	JMP	#C072
C06B	38			SEC	
C06C	A5	FE		LDA	\$FE
C06E	E9	03		SBC	##03
C070	85	FE		STA	\$FE
C072	A5	FD		LDA	\$FD
C074	C9	39		CMP	##39
C076	30	A5		BMI	#C01D
C078	78			SEI	
C079	A9	90		LDA	##90
C07B	8D	14	03	STA	\$0314
C07E	A9	C0		LDA	##C0
C080	8D	15	03	STA	\$0315
C083	58			CLI	
C084	60			RTS	
C085	A6	FB		LDX	\$FB
C087	E0	FF		CPX	##FF
C089	D0	02		BNE	#C08D
C08B	E6	FC		INC	\$FC
C08D	E6	FB		INC	\$FB
C08F	60			RTS	
C090	A5	C5		LDA	\$C5
C092	C5	FE		CMP	\$FE
C094	F0	3A		BEQ	#C0D0
C096	C9	03		CMP	##03
C098	30	36		BMI	#C0D0
C09A	C9	07		CMP	##07
C09C	10	32		BPL	#C0D0
C09E	85	FE		STA	\$FE
C0A0	C9	03		CMP	##03
C0A2	D0	03		BNE	#C0A7
C0A4	18			CLC	
C0A5	69	04		ADC	##04
C0A7	18			CLC	
C0A8	69	B1		ADC	##B1
C0AA	AE	8D	02	LDX	\$028D
C0AD	F0	03		BEQ	#C0B2
C0AF	18			CLC	
C0B0	69	04		ADC	##04
C0B2	85	FD		STA	\$FD
C0B4	A0	00		LDY	##00
C0B6	A9	C2		LDA	##C2
C0B8	85	FC		STA	\$FC
C0BA	84	FB		STY	\$FB
C0BC	B1	FB		LDA	(\$FB),Y
C0BE	C5	FD		CMP	\$FD
C0C0	F0	13		BEQ	#C0D5
C0C2	C8			INY	
C0C3	D0	F7		BNE	#C0BC
C0C5	E6	FC		INC	\$FC
C0C7	A5	FC		LDA	\$FC
C0C9	C9	C5		CMP	##C5

C0CB	D0	EF	BNE	\$C0BC	
C0CD	4C	31	EAB	JMP	\$EA31
C0D0	85	FE	STA	\$FE	
C0D2	4C	31	EAB	JMP	\$EA31
C0D5	C8		INY		
C0D6	D0	08	BNE	\$C0E0	
C0D8	E6	FC	INC	\$FC	
C0DA	A5	FC	LDA	\$FC	
C0DC	C9	C5	CMP	#\$C5	
C0DE	F0	F2	BEQ	\$C0D2	
C0E0	B1	FB	LDA	(\$FB),Y	
C0E2	C9	0D	CMP	#\$0D	
C0E4	D0	0A	BNE	\$C0F0	
C0E6	E6	C6	INC	\$C6	
C0E8	A6	C6	LDX	\$C6	
C0EA	9D	77	02	STA	\$0277,X
C0ED	4C	D5	C0	JMP	\$C0D5
C0F0	C9	00	CMP	#\$00	
C0F2	F0	DE	BEQ	\$C0D2	
C0F4	C9	85	CMP	#\$85	
C0F6	30	07	BMI	\$C0FF	
C0F8	C9	8D	CMP	#\$8D	
C0FA	10	03	BPL	\$C0FF	
C0FC	4C	31	EAB	JMP	\$EA31
C0FF	20	D2	FF	JSR	\$FFD2
C102	4C	D5	C0	JMP	\$C0D5
C105	00		BRK		

## To conclude

And that's all there is to adding commands. Just intercept the Basic routines that usually look after everything, and write your own to do the job instead.

As your knowledge of machine code grows, so can the complexity and indeed the number of the commands that you would want to add.

Perhaps one day we'll see a 'Smith's Basic' on the market for the Commodore 64!

# Appendix

## An assembler/disassembler

### Introduction

A familiar little program, presented here for the first time in disassembled form, so that you can get a better idea of how it works, and also make it a lot easier to type in, if you choose to undertake the mammoth task involved.

The monitor can be accessed at any time with a SYS 2176 command, but if you want to move it to somewhere else in memory, just use the Transfer Memory feature and put Extramon anywhere you like.

Make a note of where Extramon now lives, if you do move it, so that you'll know the correct SYS call to give.

And now, it's time to get the fingers tapping!

B\*

	PC	SR	AC	XR	YR	SP	
.	:	87D8	33	00	C0	00	F6
.							
0880	A5	2D					LDA \$2D
0882	85	22					STA \$22
0884	A5	2E					LDA \$2E
0886	85	23					STA \$23
0888	A5	37					LDA \$37
088A	85	24					STA \$24
088C	A5	38					LDA \$38
088E	85	25					STA \$25
0890	A0	00					LDY #\$00
0892	A5	22					LDA \$22
0894	D0	02					BNE \$0898
0896	C6	23					DEC \$23
0898	C6	22					DEC \$22
089A	B1	22					LDA (\$22),Y
089C	D0	3C					BNE \$08DA
089E	A5	22					LDA \$22
08A0	D0	02					BNE \$08A4
08A2	C6	23					DEC \$23
08A4	C6	22					DEC \$22
08A6	B1	22					LDA (\$22),Y
08AB	F0	21					BEQ \$08CB
08AA	85	26					STA \$26
08AC	A5	22					LDA \$22
08AE	D0	02					BNE \$08B2
08B0	C6	23					DEC \$23
08B2	C6	22					DEC \$22
08B4	B1	22					LDA (\$22),Y
08B6	18						CLC
08B7	65	24					ADC \$24
08B9	AA						TAX
08BA	A5	26					LDA \$26
08BC	65	25					ADC \$25
08BE	48						PHA
08BF	A5	37					LDA \$37
08C1	D0	02					BNE \$08C5
08C3	C6	38					DEC \$38
08C5	C6	37					DEC \$37
08C7	68						PLA
08C8	91	37					STA (\$37),Y
08CA	8A						TXA
08CB	48						PHA
08CC	A5	37					LDA \$37
08CE	D0	02					BNE \$08D2
08D0	C6	38					DEC \$38
08D2	C6	37					DEC \$37
08D4	68						PLA
08D5	91	37					STA (\$37),Y
08D7	18						CLC



08DB	90	B6	BCC	\$0890
08DA	C9	4F	CMP	##4F
08DC	D0	ED	BNE	\$08CB
08DE	A5	37	LDA	\$37
08E0	85	33	STA	\$33
08E2	A5	38	LDA	\$38
08E4	85	34	STA	\$34
08E6	6C	37 00	JMP	(\$0037)
08E9	4F		???	
08EA	4F		???	
08EB	4F		???	
08EC	4F		???	
08ED	AD	E6 FF	LDA	\$FFE6
08F0	00		BRK	
08F1	8D	16 03	STA	\$0316
08F4	AD	E7 FF	LDA	\$FFE7
08F7	00		BRK	
08F8	8D	17 03	STA	\$0317
08FB	A9	80	LDA	##80
08FD	20	90 FF	JSR	\$FF90
0900	00		BRK	
0901	00		BRK	
0902	D8		CLD	
0903	68		PLA	
0904	8D	3E 02	STA	\$023E
0907	68		PLA	
0908	8D	3D 02	STA	\$023D
090B	68		PLA	
090C	8D	3C 02	STA	\$023C
090F	68		PLA	
0910	8D	3B 02	STA	\$023B
0913	68		PLA	
0914	AA		TAX	
0915	68		PLA	
0916	AB		TAY	
0917	38		SEC	
0918	8A		TXA	
0919	E9	02	SBC	##02
091B	8D	3A 02	STA	\$023A
091E	98		TYA	
091F	E9	00	SBC	##00
0921	00		BRK	
0922	8D	39 02	STA	\$0239
0925	BA		TSX	
0926	8E	3F 02	STX	\$023F
0929	20	57 FD	JSR	\$FD57
092C	00		BRK	
092D	A2	42	LDX	##42
092F	A9	2A	LDA	##2A
0931	20	57 FA	JSR	\$FA57
0934	00		BRK	
0935	A9	52	LDA	##52

0937	D0	34	BNE	\$096D
0939	E6	C1	INC	\$C1
093B	D0	06	BNE	\$0943
093D	E6	C2	INC	\$C2
093F	D0	02	BNE	\$0943
0941	E6	26	INC	\$26
0943	60		RTS	
0944	20	CF FF	JSR	\$FFCF
0947	C9	0D	CMP	##0D
0949	D0	F8	BNE	\$0943
094B	68		PLA	
094C	68		PLA	
094D	A9	90	LDA	##90
094F	20	D2 FF	JSR	\$FFD2
0952	A9	00	LDA	##00
0954	00		BRK	
0955	85	26	STA	\$26
0957	A2	0D	LDX	##0D
0959	A9	2E	LDA	##2E
095B	20	57 FA	JSR	\$FA57
095E	00		BRK	
095F	A9	05	LDA	##05
0961	20	D2 FF	JSR	\$FFD2
0964	20	3E F8	JSR	\$F83E
0967	00		BRK	
0968	C9	2E	CMP	##2E
096A	F0	F9	BEQ	\$0965
096C	C9	20	CMP	##20
096E	F0	F5	BEQ	\$0965
0970	A2	0E	LDX	##0E
0972	DD	B7 FF	CMP	\$FFB7,X
0975	00		BRK	
0976	D0	0C	BNE	\$0984
0978	8A		TXA	
0979	0A		ASL	
097A	AA		TAX	
097B	BD	C7 FF	LDA	\$FFC7,X
097E	00		BRK	
097F	48		PHA	
0980	BD	C6 FF	LDA	\$FFC6,X
0983	00		BRK	
0984	48		PHA	
0985	60		RTS	
0986	CA		DEX	
0987	10	EC	BPL	\$0975
0989	4C	ED FA	JMP	\$FAED
098C	00		BRK	
098D	A5	C1	LDA	\$C1
098F	BD	3A 02	STA	\$023A
0992	A5	C2	LDA	\$C2
0994	BD	39 02	STA	\$0239
0997	60		RTS	

0998	A9	08	LDA #\$08
099A	B5	1D	STA \$1D
099C	A0	00	LDY #\$00
099E	00		BRK
099F	20	54 FD	JSR \$FD54
09A2	00		BRK
09A3	B1	C1	LDA (\$C1),Y
09A5	20	48 FA	JSR \$FA48
09A8	00		BRK
09A9	20	33 F8	JSR \$F833
09AC	00		BRK
09AD	C6	1D	DEC \$1D
09AF	D0	F1	BNE \$09A2
09B1	60		RTS
09B2	20	88 FA	JSR \$FA88
09B5	00		BRK
09B6	90	0B	BCC \$09C3
09B8	A2	00	LDX #\$00
09BA	00		BRK
09BB	B1	C1	STA (\$C1,X)
09BD	C1	C1	CMP (\$C1,X)
09BF	F0	03	BEQ \$09C4
09C1	4C	ED FA	JMP \$FAED
09C4	00		BRK
09C5	20	33 F8	JSR \$F833
09C8	00		BRK
09C9	C6	1D	DEC \$1D
09CB	60		RTS
09CC	A9	3B	LDA #\$3B
09CE	B5	C1	STA \$C1
09D0	A9	02	LDA #\$02
09D2	B5	C2	STA \$C2
09D4	A9	05	LDA #\$05
09D6	60		RTS
09D7	9B		TYA
09D8	4B		PHA
09D9	20	57 FD	JSR \$FD57
09DC	00		BRK
09DD	6B		PLA
09DE	A2	2E	LDX #\$2E
09E0	4C	57 FA	JMP \$FA57
09E3	00		BRK
09E4	A9	90	LDA #\$90
09E6	20	D2 FF	JSR \$FFD2
09E9	A2	00	LDX #\$00
09EB	00		BRK
09EC	BD	EA FF	LDA \$FFEA,X
09EF	00		BRK
09F0	20	D2 FF	JSR \$FFD2
09F3	E8		INX
09F4	E0	16	CPX #\$16
09F6	D0	F5	BNE \$09ED

09F8	A0	3B	LDY	##3B
09FA	20	C2 F8	JSR	\$F8C2
09FD	00		BRK	
09FE	AD	39 02	LDA	\$0239
0A01	20	48 FA	JSR	\$FA48
0A04	00		BRK	
0A05	AD	3A 02	LDA	\$023A
0A08	20	48 FA	JSR	\$FA48
0A0B	00		BRK	
0A0C	20	B7 F8	JSR	\$F8B7
0A0F	00		BRK	
0A10	20	8D F8	JSR	\$F88D
0A13	00		BRK	
0A14	F0	5C	BEQ	\$0A72
0A16	20	3E F8	JSR	\$F83E
0A19	00		BRK	
0A1A	20	79 FA	JSR	\$FA79
0A1D	00		BRK	
0A1E	90	33	BCC	\$0A53
0A20	20	69 FA	JSR	\$FA69
0A23	00		BRK	
0A24	20	3E F8	JSR	\$F83E
0A27	00		BRK	
0A28	20	79 FA	JSR	\$FA79
0A2B	00		BRK	
0A2C	90	2B	BCC	\$0A56
0A2E	20	69 FA	JSR	\$FA69
0A31	00		BRK	
0A32	A9	90	LDA	##90
0A34	20	D2 FF	JSR	\$FFD2
0A37	20	E1 FF	JSR	\$FFE1
0A3A	F0	3C	BEQ	\$0A78
0A3C	A6	26	LDX	\$26
0A3E	D0	3B	BNE	\$0A78
0A40	A5	C3	LDA	\$C3
0A42	C5	C1	CMP	\$C1
0A44	A5	C4	LDA	\$C4
0A46	E5	C2	SBC	\$C2
0A48	90	2E	BCC	\$0A78
0A4A	A0	3A	LDY	##3A
0A4C	20	C2 F8	JSR	\$F8C2
0A4F	00		BRK	
0A50	20	41 FA	JSR	\$FA41
0A53	00		BRK	
0A54	20	8B F8	JSR	\$F88B
0A57	00		BRK	
0A58	F0	E0	BEQ	\$0A3A
0A5A	4C	ED FA	JMP	\$FAED
0A5D	00		BRK	
0A5E	20	79 FA	JSR	\$FA79
0A61	00		BRK	
0A62	90	03	BCC	\$0A67

0A64	20	80	F8	JSR	\$F880
0A67	00			BRK	
0A68	20	B7	F8	JSR	\$F8B7
0A6B	00			BRK	
0A6C	D0	07		BNE	\$0A75
0A6E	20	79	FA	JSR	\$FA79
0A71	00			BRK	
0A72	90	EB		BCC	\$0A5F
0A74	A9	08		LDA	#\$08
0A76	85	1D		STA	\$1D
0A78	20	3E	F8	JSR	\$F83E
0A7B	00			BRK	
0A7C	20	A1	F8	JSR	\$FBA1
0A7F	00			BRK	
0A80	D0	F8		BNE	\$0A7A
0A82	4C	47	F8	JMP	\$F847
0A85	00			BRK	
0A86	20	CF	FF	JSR	\$FFCF
0A89	C9	0D		CMP	#\$0D
0A8B	F0	0C		BEQ	\$0A99
0A8D	C9	20		CMP	#\$20
0A8F	D0	D1		BNE	\$0A62
0A91	20	79	FA	JSR	\$FA79
0A94	00			BRK	
0A95	90	03		BCC	\$0A9A
0A97	20	80	F8	JSR	\$F880
0A9A	00			BRK	
0A9B	A9	90		LDA	#\$90
0A9D	20	D2	FF	JSR	\$FFD2
0AA0	AE	3F	02	LDX	\$023F
0AA3	9A			TXS	
0AA4	7B			SEI	
0AA5	AD	39	02	LDA	\$0239
0AAB	4B			PHA	
0AA9	AD	3A	02	LDA	\$023A
0AAC	4B			PHA	
0AAD	AD	3B	02	LDA	\$023B
0AB0	4B			PHA	
0AB1	AD	3C	02	LDA	\$023C
0AB4	AE	3D	02	LDX	\$023D
0AB7	AC	3E	02	LDY	\$023E
0ABA	40			RTI	
0ABB	A9	90		LDA	#\$90
0ABD	20	D2	FF	JSR	\$FFD2
0AC0	AE	3F	02	LDX	\$023F
0AC3	9A			TXS	
0AC4	6C	02	A0	JMP	(\$A002)
0AC7	A0	01		LDY	#\$01
0AC9	84	BA		STY	\$BA
0ACB	84	B9		STY	\$B9
0ACD	8B			DEY	
0ACE	84	B7		STY	\$B7

0AD0	B4	90	STY	\$90
0AD2	B4	93	STY	\$93
0AD4	A9	40	LDA	#\$40
0AD6	B5	BB	STA	\$BB
0AD8	A9	02	LDA	#\$02
0ADA	B5	BC	STA	\$BC
0ADC	20	CF FF	JSR	\$FFCF
0ADF	C9	20	CMP	#\$20
0AE1	F0	F9	BEQ	\$0ADC
0AE3	C9	0D	CMP	#\$0D
0AE5	F0	38	BEQ	\$0B1F
0AE7	C9	22	CMP	#\$22
0AE9	D0	14	BNE	\$0AFF
0AEB	20	CF FF	JSR	\$FFCF
0AEE	C9	22	CMP	#\$22
0AF0	F0	10	BEQ	\$0B02
0AF2	C9	0D	CMP	#\$0D
0AF4	F0	29	BEQ	\$0B1F
0AF6	91	BB	STA	(\$BB),Y
0AF8	E6	B7	INC	\$B7
0AFA	CB		INY	
0AFB	C0	10	CPY	##10
0AFD	D0	EC	BNE	\$0AEB
0AFF	4C	ED FA	JMP	\$FAED
0B02	00		BRK	
0B03	20	CF FF	JSR	\$FFCF
0B06	C9	0D	CMP	#\$0D
0B08	F0	16	BEQ	\$0B20
0B0A	C9	2C	CMP	#\$2C
0B0C	D0	DC	BNE	\$0AEA
0B0E	20	BB FA	JSR	\$FABB
0B11	00		BRK	
0B12	29	0F	AND	#\$0F
0B14	F0	E9	BEQ	\$0AFF
0B16	C9	03	CMP	#\$03
0B18	F0	E5	BEQ	\$0AFF
0B1A	B5	BA	STA	\$BA
0B1C	20	CF FF	JSR	\$FFCF
0B1F	C9	0D	CMP	#\$0D
0B21	60		RTS	
0B22	6C	30 03	JMP	(\$0330)
0B25	6C	32 03	JMP	(\$0332)
0B28	20	96 F9	JSR	\$F996
0B2B	00		BRK	
0B2C	D0	D4	BNE	\$0B02
0B2E	A9	90	LDA	#\$90
0B30	20	D2 FF	JSR	\$FFD2
0B33	A9	00	LDA	#\$00
0B35	00		BRK	
0B36	20	EF F9	JSR	\$F9EF
0B39	00		BRK	
0B3A	A5	90	LDA	\$90

0B3C	29	10	AND	##10
0B3E	D0	C4	BNE	\$0B04
0B40	4C	47 F8	JMP	\$F847
0B43	00		BRK	
0B44	20	96 F9	JSR	\$F996
0B47	00		BRK	
0B48	C9	2C	CMP	##2C
0B4A	D0	BA	BNE	\$0B06
0B4C	20	79 FA	JSR	\$FA79
0B4F	00		BRK	
0B50	20	69 FA	JSR	\$FA69
0B53	00		BRK	
0B54	20	CF FF	JSR	\$FFCF
0B57	C9	2C	CMP	##2C
0B59	D0	AD	BNE	\$0B08
0B5B	20	79 FA	JSR	\$FA79
0B5E	00		BRK	
0B5F	A5	C1	LDA	\$C1
0B61	85	AE	STA	\$AE
0B63	A5	C2	LDA	\$C2
0B65	85	AF	STA	\$AF
0B67	20	69 FA	JSR	\$FA69
0B6A	00		BRK	
0B6B	20	CF FF	JSR	\$FFCF
0B6E	C9	0D	CMP	##0D
0B70	D0	98	BNE	\$0B0A
0B72	A9	90	LDA	##90
0B74	20	D2 FF	JSR	\$FFD2
0B77	20	F2 F9	JSR	\$F9F2
0B7A	00		BRK	
0B7B	4C	47 F8	JMP	\$F847
0B7E	00		BRK	
0B7F	A5	C2	LDA	\$C2
0B81	20	48 FA	JSR	\$FA48
0B84	00		BRK	
0B85	A5	C1	LDA	\$C1
0B87	48		PHA	
0B88	4A		LSR	
0B89	4A		LSR	
0B8A	4A		LSR	
0B8B	4A		LSR	
0B8C	20	60 FA	JSR	\$FA60
0B8F	00		BRK	
0B90	AA		TAX	
0B91	68		PLA	
0B92	29	0F	AND	##0F
0B94	20	60 FA	JSR	\$FA60
0B97	00		BRK	
0B98	48		PHA	
0B99	8A		TXA	
0B9A	20	D2 FF	JSR	\$FFD2
0B9D	68		PLA	

0B9E	4C	D2	FF	JMP	\$FFD2
0BA1	09	30		ORA	#\$30
0BA3	C9	3A		CMP	#\$3A
0BA5	90	02		BCC	\$0BA9
0BA7	69	06		ADC	#\$06
0BA9	60			RTS	
0BAA	A2	02		LDX	#\$02
0BAC	B5	C0		LDA	\$C0,X
0BAE	48			PHA	
0BAF	B5	C2		LDA	\$C2,X
0BB1	95	C0		STA	\$C0,X
0BB3	68			PLA	
0BB4	95	C2		STA	\$C2,X
0BB6	CA			DEX	
0BB7	D0	F3		BNE	\$0BAC
0BB9	60			RTS	
0BBA	20	88	FA	JSR	\$FAB8
0BBD	00			BRK	
0BBE	90	02		BCC	\$0BC2
0BC0	B5	C2		STA	\$C2
0BC2	20	88	FA	JSR	\$FAB8
0BC5	00			BRK	
0BC6	90	02		BCC	\$0BCA
0BC8	B5	C1		STA	\$C1
0BCA	60			RTS	
0BCB	A9	00		LDA	#\$00
0BCD	00			BRK	
0BCE	B5	2A		STA	\$2A
0BD0	20	3E	F8	JSR	\$FB3E
0BD3	00			BRK	
0BD4	C9	20		CMP	#\$20
0BD6	D0	09		BNE	\$0BE1
0BD8	20	3E	F8	JSR	\$FB3E
0BDB	00			BRK	
0BDC	C9	20		CMP	#\$20
0BDE	D0	0E		BNE	\$0BEE
0BE0	18			CLC	
0BE1	60			RTS	
0BE2	20	AF	FA	JSR	\$FAAF
0BE5	00			BRK	
0BE6	0A			ASL	
0BE7	0A			ASL	
0BE8	0A			ASL	
0BE9	0A			ASL	
0BEA	B5	2A		STA	\$2A
0BEC	20	3E	F8	JSR	\$FB3E
0BEF	00			BRK	
0BF0	20	AF	FA	JSR	\$FAAF
0BF3	00			BRK	
0BF4	05	2A		ORA	\$2A
0BF6	38			SEC	
0BF7	60			RTS	



0BF8	C9	3A		CMP	##3A
0BFA	90	02		BCC	\$0BFE
0BFC	69	08		ADC	##08
0BFE	29	0F		AND	##0F
0C00	60			RTS	
0C01	A2	02		LDX	##02
0C03	2C	A2	00	BIT	\$00A2
0C06	00			BRK	
0C07	B4	C1		LDY	\$C1,X
0C09	D0	08		BNE	\$0C13
0C0B	B4	C2		LDY	\$C2,X
0C0D	D0	02		BNE	\$0C11
0C0F	E6	26		INC	\$26
0C11	D6	C2		DEC	\$C2,X
0C13	D6	C1		DEC	\$C1,X
0C15	60			RTS	
0C16	20	3E	F8	JSR	\$FB3E
0C19	00			BRK	
0C1A	C9	20		CMP	##20
0C1C	F0	F9		BEQ	\$0C17
0C1E	60			RTS	
0C1F	A9	00		LDA	##00
0C21	00			BRK	
0C22	8D	00	00	STA	\$0000
0C25	01	20		ORA	(\$20,X)
0C27	CC	FA	00	CPY	\$00FA
0C2A	20	8F	FA	JSR	\$FABF
0C2D	00			BRK	
0C2E	20	7C	FA	JSR	\$FA7C
0C31	00			BRK	
0C32	90	09		BCC	\$0C3D
0C34	60			RTS	
0C35	20	3E	F8	JSR	\$FB3E
0C38	00			BRK	
0C39	20	79	FA	JSR	\$FA79
0C3C	00			BRK	
0C3D	B0	DE		BCS	\$0C1D
0C3F	AE	3F	02	LDX	\$023F
0C42	9A			TXS	
0C43	A9	90		LDA	##90
0C45	20	D2	FF	JSR	\$FFD2
0C48	A9	3F		LDA	##3F
0C4A	20	D2	FF	JSR	\$FFD2
0C4D	4C	47	F8	JMP	\$FB47
0C50	00			BRK	
0C51	20	54	FD	JSR	\$FD54
0C54	00			BRK	
0C55	CA			DEX	
0C56	D0	FA		BNE	\$0C52
0C58	60			RTS	
0C59	E6	C3		INC	\$C3
0C5B	D0	02		BNE	\$0C5F

0C5D	E6	C4	INC	\$C4
0C5F	60		RTS	
0C60	A2	02	LDX	#\$02
0C62	B5	C0	LDA	\$C0,X
0C64	48		PHA	
0C65	B5	27	LDA	\$27,X
0C67	95	C0	STA	\$C0,X
0C69	68		PLA	
0C6A	95	27	STA	\$27,X
0C6C	CA		DEX	
0C6D	D0	F3	BNE	\$0C62
0C6F	60		RTS	
0C70	A5	C3	LDA	\$C3
0C72	A4	C4	LDY	\$C4
0C74	38		SEC	
0C75	E9	02	SBC	#\$02
0C77	B0	0E	BCS	\$0CB7
0C79	88		DEY	
0C7A	90	0B	BCC	\$0CB7
0C7C	A5	28	LDA	\$28
0C7E	A4	29	LDY	\$29
0C80	4C	33 FB	JMP	\$FB33
0C83	00		BRK	
0C84	A5	C3	LDA	\$C3
0C86	A4	C4	LDY	\$C4
0C88	38		SEC	
0C89	E5	C1	SBC	\$C1
0C8B	85	1E	STA	\$1E
0C8D	98		TYA	
0C8E	E5	C2	SBC	\$C2
0C90	A8		TAY	
0C91	05	1E	ORA	\$1E
0C93	60		RTS	
0C94	20	D4 FA	JSR	\$FAD4
0C97	00		BRK	
0C98	20	69 FA	JSR	\$FA69
0C9B	00		BRK	
0C9C	20	E5 FA	JSR	\$FAE5
0C9F	00		BRK	
0CA0	20	0C FB	JSR	\$FB0C
0CA3	00		BRK	
0CA4	20	E5 FA	JSR	\$FAE5
0CA7	00		BRK	
0CAB	20	2F FB	JSR	\$FB2F
0CAB	00		BRK	
0CAC	20	69 FA	JSR	\$FA69
0CAF	00		BRK	
0CB0	90	15	BCC	\$0CC7
0CB2	A6	26	LDX	\$26
0CB4	D0	64	BNE	\$0D1A
0CB6	20	28 FB	JSR	\$FB28
0CB9	00		BRK	

OCBA	90	5F	BCC	\$0D1B
OCBC	A1	C1	LDA	(\$C1,X)
OCBE	81	C3	STA	(\$C3,X)
OCC0	20	05 FB	JSR	\$FB05
OCC3	00		BRK	
OCC4	20	33 FB	JSR	\$FB33
OCC7	00		BRK	
OCC8	D0	EB	BNE	\$0CB5
OCCA	20	28 FB	JSR	\$FB28
OCCD	00		BRK	
OCCE	18		CLC	
OCCF	A5	1E	LDA	\$1E
OCD1	65	C3	ADC	\$C3
OCD3	85	C3	STA	\$C3
OCD5	98		TYA	
OCD6	65	C4	ADC	\$C4
OCD8	85	C4	STA	\$C4
OCDA	20	0C FB	JSR	\$FB0C
OCDD	00		BRK	
OCDE	A6	26	LDX	\$26
OCE0	D0	3D	BNE	\$0D1F
OCE2	A1	C1	LDA	(\$C1,X)
OCE4	81	C3	STA	(\$C3,X)
OCE6	20	28 FB	JSR	\$FB28
OCE9	00		BRK	
OCEA	B0	34	BCS	\$0D20
OCEC	20	8B FA	JSR	\$FABB
OCEF	00		BRK	
OCF0	20	8B FA	JSR	\$FABB
OCF3	00		BRK	
OCF4	4C	7D FB	JMP	\$FB7D
OCF7	00		BRK	
OCF8	20	D4 FA	JSR	\$FAD4
OCFB	00		BRK	
OCFC	20	69 FA	JSR	\$FA69
OCFF	00		BRK	
OD00	20	E5 FA	JSR	\$FAE5
OD03	00		BRK	
OD04	20	69 FA	JSR	\$FA69
OD07	00		BRK	
OD08	20	3E FB	JSR	\$FB3E
OD0B	00		BRK	
OD0C	20	8B FA	JSR	\$FABB
OD0F	00		BRK	
OD10	90	14	BCC	\$0D26
OD12	85	1D	STA	\$1D
OD14	A6	26	LDX	\$26
OD16	D0	11	BNE	\$0D29
OD18	20	2F FB	JSR	\$FB2F
OD1B	00		BRK	
OD1C	90	0C	BCC	\$0D2A
OD1E	A5	1D	LDA	\$1D

0D20	81	C1		STA (\$C1,X)
0D22	20	33	F8	JSR \$F833
0D25	00			BRK
0D26	D0	EE		BNE \$0D16
0D28	4C	ED	FA	JMP \$FAED
0D2B	00			BRK
0D2C	4C	47	F8	JMP \$F847
0D2F	00			BRK
0D30	20	D4	FA	JSR \$FAD4
0D33	00			BRK
0D34	20	69	FA	JSR \$FA69
0D37	00			BRK
0D38	20	E5	FA	JSR \$FAE5
0D3B	00			BRK
0D3C	20	69	FA	JSR \$FA69
0D3F	00			BRK
0D40	20	3E	F8	JSR \$F83E
0D43	00			BRK
0D44	A2	00		LDX #\$00
0D46	00			BRK
0D47	20	3E	F8	JSR \$F83E
0D4A	00			BRK
0D4B	C9	27		CMP #\$27
0D4D	D0	14		BNE \$0D63
0D4F	20	3E	F8	JSR \$F83E
0D52	00			BRK
0D53	9D	10	02	STA \$0210,X
0D56	E8			INX
0D57	20	CF	FF	JSR \$FFCF
0D5A	C9	0D		CMP #\$0D
0D5C	F0	22		BEQ \$0D80
0D5E	E0	20		CPX #\$20
0D60	D0	F1		BNE \$0D53
0D62	F0	1C		BEQ \$0D80
0D64	8E	00	00	STX \$0000
0D67	01	20		DRA (\$20,X)
0D69	BF			???
0D6A	FA			???
0D6B	00			BRK
0D6C	90	C6		BCC \$0D34
0D6E	9D	10	02	STA \$0210,X
0D71	E8			INX
0D72	20	CF	FF	JSR \$FFCF
0D75	C9	0D		CMP #\$0D
0D77	F0	09		BEQ \$0D82
0D79	20	88	FA	JSR \$FAB8
0D7C	00			BRK
0D7D	90	B6		BCC \$0D35
0D7F	E0	20		CPX #\$20
0D81	D0	EC		BNE \$0D6F
0D83	86	1C		STX \$1C
0D85	A9	90		LDA #\$90

OD87	20	D2	FF	JSR	\$FFD2
OD8A	20	57	FD	JSR	\$FD57
OD8D	00			BRK	
OD8E	A2	00		LDX	#\$00
OD90	00			BRK	
OD91	A0	00		LDY	#\$00
OD93	00			BRK	
OD94	B1	C1		LDA	(\$C1),Y
OD96	DD	10	02	CMP	\$0210,X
OD99	D0	0C		BNE	\$0DA7
OD9B	CB			INY	
OD9C	EB			INX	
OD9D	E4	1C		CPX	\$1C
OD9F	D0	F3		BNE	\$0D94
ODA1	20	41	FA	JSR	\$FA41
ODA4	00			BRK	
ODA5	20	54	FD	JSR	\$FD54
ODAB	00			BRK	
ODA9	20	33	FB	JSR	\$FB33
ODAC	00			BRK	
ODAD	A6	26		LDX	\$26
ODAF	D0	8D		BNE	\$0D3E
ODB1	20	2F	FB	JSR	\$FB2F
ODB4	00			BRK	
ODB5	B0	DD		BCS	\$0D94
ODB7	4C	47	FB	JMP	\$FB47
ODBA	00			BRK	
ODBB	20	D4	FA	JSR	\$FAD4
ODBE	00			BRK	
ODBF	85	20		STA	\$20
ODC1	A5	C2		LDA	\$C2
ODC3	85	21		STA	\$21
ODC5	A2	00		LDX	#\$00
ODC7	00			BRK	
ODC8	86	28		STX	\$28
ODCA	A9	93		LDA	#\$93
ODCC	20	D2	FF	JSR	\$FFD2
ODCF	A9	90		LDA	#\$90
ODD1	20	D2	FF	JSR	\$FFD2
ODD4	A9	16		LDA	#\$16
ODD6	85	1D		STA	\$1D
ODD8	20	6A	FC	JSR	\$FC6A
ODDB	00			BRK	
ODDC	20	CA	FC	JSR	\$FCCA
ODDF	00			BRK	
ODE0	85	C1		STA	\$C1
ODE2	84	C2		STY	\$C2
ODE4	C6	1D		DEC	\$1D
ODE6	D0	F2		BNE	\$0DDA
ODE8	A9	91		LDA	#\$91
ODEA	20	D2	FF	JSR	\$FFD2
ODED	4C	47	FB	JMP	\$FB47

ODF0	00		BRK
ODF1	A0	2C	LDY #\$2C
ODF3	20	C2 FB	JSR \$F8C2
ODF6	00		BRK
ODF7	20	54 FD	JSR \$FD54
ODFA	00		BRK
ODFB	20	41 FA	JSR \$FA41
ODFE	00		BRK
ODFF	20	54 FD	JSR \$FD54
OE02	00		BRK
OE03	A2	00	LDX #\$00
OE05	00		BRK
OE06	A1	C1	LDA (\$C1,X)
OE08	20	D9 FC	JSR \$FCD9
OE0B	00		BRK
OE0C	4B		PHA
OE0D	20	1F FD	JSR \$FD1F
OE10	00		BRK
OE11	6B		PLA
OE12	20	35 FD	JSR \$FD35
OE15	00		BRK
OE16	A2	06	LDX #\$06
OE18	E0	03	CPX #\$03
OE1A	D0	12	BNE \$0E2E
OE1C	A4	1F	LDY \$1F
OE1E	F0	0E	BEQ \$0E2E
OE20	A5	2A	LDA \$2A
OE22	C9	E8	CMP #\$E8
OE24	B1	C1	LDA (\$C1),Y
OE26	B0	1C	BCS \$0E44
OE2B	20	C2 FC	JSR \$FCC2
OE2B	00		BRK
OE2C	8B		DEY
OE2D	D0	F2	BNE \$0E21
OE2F	06	2A	ASL \$2A
OE31	90	0E	BCC \$0E41
OE33	BD	2A FF	LDA \$FF2A,X
OE36	00		BRK
OE37	20	A5 FD	JSR \$FDA5
OE3A	00		BRK
OE3B	BD	30 FF	LDA \$FF30,X
OE3E	00		BRK
OE3F	F0	03	BEQ \$0E44
OE41	20	A5 FD	JSR \$FDA5
OE44	00		BRK
OE45	CA		DEX
OE46	D0	D5	BNE \$0E1D
OE4B	60		RTS
OE49	20	CD FC	JSR \$FCCD
OE4C	00		BRK
OE4D	AA		TAX
OE4E	EB		INX

0E4F	D0	01		BNE	\$0E52
0E51	C8			INY	
0E52	98			TYA	
0E53	20	C2	FC	JSR	\$FCC2
0E56	00			BRK	
0E57	8A			TXA	
0E58	86	1C		STX	\$1C
0E5A	20	48	FA	JSR	\$FA48
0E5D	00			BRK	
0E5E	A6	1C		LDX	\$1C
0E60	60			RTS	
0E61	A5	1F		LDA	\$1F
0E63	38			SEC	
0E64	A4	C2		LDY	\$C2
0E66	AA			TAX	
0E67	10	01		BPL	\$0E6A
0E69	88			DEY	
0E6A	65	C1		ADC	\$C1
0E6C	90	01		BCC	\$0E6F
0E6E	C8			INY	
0E6F	60			RTS	
0E70	A8			TAY	
0E71	4A			LSR	
0E72	90	0B		BCC	\$0E7F
0E74	4A			LSR	
0E75	B0	17		BCS	\$0E8E
0E77	C9	22		CMP	#\$22
0E79	F0	13		BEQ	\$0E8E
0E7B	29	07		AND	#\$07
0E7D	09	80		ORA	#\$80
0E7F	4A			LSR	
0E80	AA			TAX	
0E81	BD	D9	FE	LDA	\$FED9,X
0E84	00			BRK	
0E85	B0	04		BCS	\$0E8B
0E87	4A			LSR	
0E88	4A			LSR	
0E89	4A			LSR	
0E8A	4A			LSR	
0E8B	29	0F		AND	#\$0F
0E8D	D0	04		BNE	\$0E93
0E8F	A0	80		LDY	#\$80
0E91	A9	00		LDA	#\$00
0E93	00			BRK	
0E94	AA			TAX	
0E95	BD	1D	FF	LDA	\$FF1D,X
0E98	00			BRK	
0E99	85	2A		STA	\$2A
0E9B	29	03		AND	#\$03
0E9D	85	1F		STA	\$1F
0E9F	98			TYA	
0EA0	29	8F		AND	#\$8F

0EA2	AA		TAX
0EA3	9B		TYA
0EA4	A0	03	LDY #\$03
0EA6	E0	8A	CPX #\$8A
0EAB	F0	0B	BEQ \$0EB5
0EAA	4A		LSR
0EAB	90	0B	BCC \$0EB5
0EAD	4A		LSR
0EAE	4A		LSR
0EAF	09	20	DRA #\$20
0EB1	8B		DEY
0EB2	D0	FA	BNE \$0EAE
0EB4	CB		INY
0EB5	8B		DEY
0EB6	D0	F2	BNE \$0EAA
0EB8	60		RTS
0EB9	B1	C1	LDA (\$C1),Y
0EBB	20	C2 FC	JSR \$FCC2
0EBE	00		BRK
0EBF	A2	01	LDX #\$01
0EC1	20	FE FA	JSR \$FAFE
0EC4	00		BRK
0EC5	C4	1F	CPY \$1F
0EC7	CB		INY
0EC8	90	F1	BCC \$0EBB
0ECA	A2	03	LDX #\$03
0ECC	C0	04	CPY #\$04
0ECE	90	F2	BCC \$0EC2
0ED0	60		RTS
0ED1	AB		TAY
0ED2	B9	37 FF	LDA \$FF37,Y
0ED5	00		BRK
0ED6	85	2B	STA \$2B
0ED8	B9	77 FF	LDA \$FF77,Y
0EDB	00		BRK
0EDC	85	29	STA \$29
0EDE	A9	00	LDA #\$00
0EE0	00		BRK
0EE1	A0	05	LDY #\$05
0EE3	06	29	ASL \$29
0EE5	26	2B	ROL \$2B
0EE7	2A		ROL
0EE8	8B		DEY
0EE9	D0	F8	BNE \$0EE3
0EEB	69	3F	ADC #\$3F
0EED	20	D2 FF	JSR \$FFD2
0EFO	CA		DEX
0EF1	D0	EC	BNE \$0EDF
0EF3	A9	20	LDA #\$20
0EF5	2C	A9 0D	BIT \$0DA9
0EF8	4C	D2 FF	JMP \$FFD2
0EFB	20	D4 FA	JSR \$FAD4



0EFE	00			BRK
0EFF	20	69	FA	JSR \$FA69
0F02	00			BRK
0F03	20	E5	FA	JSR \$FAE5
0F06	00			BRK
0F07	20	69	FA	JSR \$FA69
0F0A	00			BRK
0F0B	A2	00		LDX #\$00
0F0D	00			BRK
0F0E	86	28		STX \$28
0F10	A9	90		LDA #\$90
0F12	20	D2	FF	JSR \$FFD2
0F15	20	57	FD	JSR \$FD57
0F18	00			BRK
0F19	20	72	FC	JSR \$FC72
0F1C	00			BRK
0F1D	20	CA	FC	JSR \$FCCA
0F20	00			BRK
0F21	85	C1		STA \$C1
0F23	84	C2		STY \$C2
0F25	20	E1	FF	JSR \$FFE1
0F28	F0	05		BEQ \$0F2F
0F2A	20	2F	FB	JSR \$FB2F
0F2D	00			BRK
0F2E	B0	E9		BCS \$0F19
0F30	4C	47	F8	JMP \$FB47
0F33	00			BRK
0F34	20	D4	FA	JSR \$FAD4
0F37	00			BRK
0F38	A9	03		LDA #\$03
0F3A	85	1D		STA \$1D
0F3C	20	3E	F8	JSR \$FB3E
0F3F	00			BRK
0F40	20	A1	F8	JSR \$F8A1
0F43	00			BRK
0F44	D0	F8		BNE \$0F3E
0F46	A5	20		LDA \$20
0F48	85	C1		STA \$C1
0F4A	A5	21		LDA \$21
0F4C	85	C2		STA \$C2
0F4E	4C	46	FC	JMP \$FC46
0F51	00			BRK
0F52	C5	28		CMP \$28
0F54	F0	03		BEQ \$0F59
0F56	20	D2	FF	JSR \$FFD2
0F59	60			RTS
0F5A	20	D4	FA	JSR \$FAD4
0F5D	00			BRK
0F5E	20	69	FA	JSR \$FA69
0F61	00			BRK
0F62	BE	11	02	STX \$0211
0F65	A2	03		LDX #\$03

0F67	20	CC	FA	JSR	\$FACC
0F6A	00			BRK	
0F6B	4B			PHA	
0F6C	CA			DEX	
0F6D	D0	F9		BNE	\$0F6B
0F6F	A2	03		LDX	##03
0F71	6B			PLA	
0F72	3B			SEC	
0F73	E9	3F		SBC	##3F
0F75	A0	05		LDY	##05
0F77	4A			LSR	
0F7B	6E	11	02	ROR	\$0211
0F7B	6E	10	02	ROR	\$0210
0F7E	BB			DEY	
0F7F	D0	F6		BNE	\$0F77
0F81	CA			DEX	
0F82	D0	ED		BNE	\$0F71
0F84	A2	02		LDX	##02
0F86	20	CF	FF	JSR	\$FFCF
0F89	C9	0D		CMP	##0D
0F8B	F0	1E		BEQ	\$0FAB
0F8D	C9	20		CMP	##20
0F8F	F0	F5		BEQ	\$0F86
0F91	20	D0	FE	JSR	\$FED0
0F94	00			BRK	
0F95	B0	0F		BCS	\$0FA6
0F97	20	9C	FA	JSR	\$FA9C
0F9A	00			BRK	
0F9B	A4	C1		LDY	\$C1
0F9D	84	C2		STY	\$C2
0F9F	85	C1		STA	\$C1
0FA1	A9	30		LDA	##30
0FA3	9D	10	02	STA	\$0210,X
0FA6	E8			INX	
0FA7	9D	10	02	STA	\$0210,X
0FAA	E8			INX	
0FAB	D0	DB		BNE	\$0F8B
0FAD	86	2B		STX	\$2B
0FAF	A2	00		LDX	##00
0FB1	00			BRK	
0FB2	86	26		STX	\$26
0FB4	F0	04		BEQ	\$0FBA
0FB6	E6	26		INC	\$26
0FB8	F0	75		BEQ	\$102F
0FBA	A2	00		LDX	##00
0FBC	00			BRK	
0FBD	86	1D		STX	\$1D
0FBF	A5	26		LDA	\$26
0FC1	20	D9	FC	JSR	\$FCD9
0FC4	00			BRK	
0FC5	A6	2A		LDX	\$2A
0FC7	86	29		STX	\$29

0FC9	AA		TAX
0FCA	BC	37 FF	LDY \$FF37,X
0FCD	00		BRK
0FCE	BD	77 FF	LDA \$FF77,X
0FD1	00		BRK
0FD2	20	B9 EE	JSR \$FEB9
0FD5	00		BRK
0FD6	D0	E3	BNE \$0FBB
0FDB	A2	06	LDX #\$06
0FDA	E0	03	CPX #\$03
0FDC	D0	19	BNE \$0FF7
0FDE	A4	1F	LDY \$1F
0FE0	F0	15	BEQ \$0FF7
0FE2	A5	2A	LDA \$2A
0FE4	C9	EB	CMP #\$EB
0FE6	A9	30	LDA #\$30
0FEB	B0	21	BCS \$100B
0FEA	20	BF FE	JSR \$FEBF
0FED	00		BRK
0FEE	D0	CC	BNE \$0FBC
OFF0	20	C1 FE	JSR \$FEC1
OFF3	00		BRK
OFF4	D0	C7	BNE \$0FBD
OFF6	88		DEY
OFF7	D0	EB	BNE \$0FE4
OFF9	06	2A	ASL \$2A
OFFB	90	0B	BCC \$100B
OFFD	BC	30 FF	LDY \$FF30,X
1000	00		BRK
1001	BD	2A FF	LDA \$FF2A,X
1004	00		BRK
1005	20	B9 FE	JSR \$FEB9
1008	00		BRK
1009	D0	B5	BNE \$0FC0
100B	CA		DEX
100C	D0	D1	BNE \$0FDF
100E	F0	0A	BEQ \$101A
1010	20	B8 FE	JSR \$FEB8
1013	00		BRK
1014	D0	AB	BNE \$0FC1
1016	20	B8 FE	JSR \$FEB8
1019	00		BRK
101A	D0	A6	BNE \$0FC2
101C	A5	28	LDA \$28
101E	C5	1D	CMP \$1D
1020	D0	A0	BNE \$0FC2
1022	20	69 FA	JSR \$FA69
1025	00		BRK
1026	A4	1F	LDY \$1F
1028	F0	28	BEQ \$1052
102A	A5	29	LDA \$29
102C	C9	9D	CMP #\$9D

102E	D0	1A		BNE	\$104A
1030	20	1C	FB	JSR	\$FB1C
1033	00			BRK	
1034	90	0A		BCC	\$1040
1036	98			TYA	
1037	D0	04		BNE	\$103D
1039	A5	1E		LDA	\$1E
103B	10	0A		BPL	\$1047
103D	4C	ED	FA	JMP	\$FAED
1040	00			BRK	
1041	CB			INY	
1042	D0	FA		BNE	\$103E
1044	A5	1E		LDA	\$1E
1046	10	F6		BPL	\$103E
1048	A4	1F		LDY	\$1F
104A	D0	03		BNE	\$104F
104C	B9	C2	00	LDA	\$00C2,Y
104F	00			BRK	
1050	91	C1		STA	(\$C1),Y
1052	88			DEY	
1053	D0	F8		BNE	\$104D
1055	A5	26		LDA	\$26
1057	91	C1		STA	(\$C1),Y
1059	20	CA	FC	JSR	\$FCCA
105C	00			BRK	
105D	85	C1		STA	\$C1
105F	84	C2		STY	\$C2
1061	A9	90		LDA	#\$90
1063	20	D2	FF	JSR	\$FFD2
1066	A0	41		LDY	#\$41
1068	20	C2	F8	JSR	\$F8C2
106B	00			BRK	
106C	20	54	FD	JSR	\$FD54
106F	00			BRK	
1070	20	41	FA	JSR	\$FA41
1073	00			BRK	
1074	20	54	FD	JSR	\$FD54
1077	00			BRK	
1078	A9	05		LDA	#\$05
107A	20	D2	FF	JSR	\$FFD2
107D	4C	B0	FD	JMP	\$FDB0
1080	00			BRK	
1081	AB			TAY	
1082	20	BF	FE	JSR	\$FEBF
1085	00			BRK	
1086	D0	11		BNE	\$1099
1088	98			TYA	
1089	F0	0E		BEQ	\$1099
108B	86	1C		STX	\$1C
108D	A6	1D		LDX	\$1D
108F	DD	10	02	CMP	\$0210,X
1092	08			PHP	

1093	EB	INX
1094	86 1D	STX \$1D
1096	A6 1C	LDX \$1C
1098	28	PLP
1099	60	RTS
109A	C9 30	CMP #\$30
109C	90 03	BCC \$10A1
109E	C9 47	CMP #\$47
10A0	60	RTS
10A1	38	SEC
10A2	60	RTS
10A3	40	RTI
10A4	02	???
10A5	45 03	EOR \$03
10A7	D0 08	BNE \$10B1
10A9	40	RTI
10AA	09 30	ORA #\$30
10AC	22	???
10AD	45 33	EOR \$33
10AF	D0 08	BNE \$10B9
10B1	40	RTI
10B2	09 40	ORA #\$40
10B4	02	???
10B5	45 33	EOR \$33
10B7	D0 08	BNE \$10C1
10B9	40	RTI
10BA	09 40	ORA #\$40
10BC	02	???
10BD	45 B3	EOR \$B3
10BF	D0 08	BNE \$10C9
10C1	40	RTI
10C2	09 00	ORA #\$00
10C4	00	BRK
10C5	22	???
10C6	44	???
10C7	33	???
10C8	D0 8C	BNE \$1056
10CA	44	???
10CB	00	BRK
10CC	00	BRK
10CD	11 22	ORA (\$22),Y
10CF	44	???
10D0	33	???
10D1	D0 8C	BNE \$105F
10D3	44	???
10D4	9A	TXS
10D5	10 22	BPL \$10F9
10D7	44	???
10D8	33	???
10D9	D0 08	BNE \$10E3
10DB	40	RTI
10DC	09 10	ORA #\$10

10DE 22	???
10DF 44	???
10E0 33	???
10E1 D0 08	BNE \$10EB
10E3 40	RTI
10E4 09 62	ORA #\$62
10E6 13	???
10E7 78	SEI
10E8 A9 00	LDA #\$00
10EA 00	BRK
10EB 21 81	AND (\$B1,X)
10ED 82	???
10EE 00	BRK
10EF 00	BRK
10F0 00	BRK
10F1 00	BRK
10F2 59 4D 91	EOR \$914D,Y
10F5 92	???
10F6 86 4A	STX \$4A
10F8 85 9D	STA \$9D
10FA 2C 29 2C	BIT \$2C29
10FD 23	???
10FE 28	PLP
10FF 24 59	BIT \$59
1101 00	BRK
1102 00	BRK
1103 58	CLI
1104 24 24	BIT \$24
1106 00	BRK
1107 00	BRK
1108 1C	???
1109 8A	TXA
110A 1C	???
110B 23	???
110C 5D 8B 1B	EOR \$1B8B,X
110F A1 9D	LDA (\$9D,X)
1111 8A	TXA
1112 1D 23 9D	ORA \$9D23,X
1115 8B	???
1116 1D A1 00	ORA \$00A1,X
1119 00	BRK
111A 29 19	AND #\$19
111C AE 69 AB	LDX \$AB69
111F 19 23 24	ORA \$2423,Y
1122 53	???
1123 1B	???
1124 23	???
1125 24 53	BIT \$53
1127 19 A1 00	ORA \$00A1,Y
112A 00	BRK
112B 1A	???
112C 5B	???

112D	5B	???
112E	A5 69	LDA \$69
1130	24 24	BIT \$24
1132	AE AE A8	LDX \$ABAE
1135	AD 29 00	LDA \$0029
1138	00	BRK
1139	7C	???
113A	00	BRK
113B	00	BRK
113C	15 9C	ORA \$9C,X
113E	6D 9C A5	ADC \$A59C
1141	69 29	ADC #\$29
1143	53	???
1144	84 13	STY \$13
1146	34	???
1147	11 A5	ORA (\$A5),Y
1149	69 23	ADC #\$23
114B	A0 DB	LDY #\$DB
114D	62	???
114E	5A	???
114F	48	PHA
1150	26 62	ROL \$62
1152	94 88	STY \$88,X
1154	54	???
1155	44	???
1156	C8	INY
1157	54	???
1158	68	PLA
1159	44	???
115A	E8	INX
115B	94 00	STY \$00,X
115D	00	BRK
115E	B4 08	LDY \$08,X
1160	84 74	STY \$74
1162	B4 28	LDY \$28,X
1164	6E 74 F4	ROR \$F474
1167	CC 4A 72	CPY \$724A
116A	F2	???
116B	A4 8A	LDY \$8A
116D	00	BRK
116E	00	BRK
116F	AA	TAX
1170	A2 A2	LDX #\$A2
1172	74	???
1173	74	???
1174	74	???
1175	72	???
1176	44	???
1177	68	PLA
1178	B2	???
1179	32	???
117A	B2	???

117B	00		BRK
117C	00		BRK
117D	22		???
117E	00		BRK
117F	00		BRK
1180	1A		???
1181	1A		???
1182	26	26	ROL \$26
1184	72		???
1185	72		???
1186	88		DEY
1187	C8		INY
1188	C4	CA	CPY \$CA
118A	26	48	ROL \$48
118C	44		???
118D	44		???
118E	A2	C8	LDX #\$C8
1190	3A		???
1191	3B		???
1192	52		???
1193	4D	47 58	EOR \$5847
1196	4C	53 54	JMP \$5453
1199	46	48	LSR \$48
119B	44		???
119C	50	2C	BVC \$11CA
119E	41	42	EOR (\$42,X)
11A0	F9	00 35	SBC \$3500,Y
11A3	F9	00 CC	SBC \$CC00,Y
11A6	F8		SED
11A7	00		BRK
11A8	F7		???
11A9	F8		SED
11AA	00		BRK
11AB	56	F9	LSR \$F9,X
11AD	00		BRK
11AE	89		???
11AF	F9	00 F4	SBC \$F400,Y
11B2	F9	00 0C	SBC \$0C00,Y
11B5	FA		???
11B6	00		BRK
11B7	3E	FB 00	ROL \$00FB,X
11BA	92		???
11BB	FB		???
11BC	00		BRK
11BD	C0	FB	CPY #\$FB
11BF	00		BRK
11C0	38		SEC
11C1	FC		???
11C2	00		BRK
11C3	5B		???
11C4	FD	00 8A	SBC \$8A00,X
11C7	FD	00 AC	SBC \$AC00,X



11CA	FD	00	46	SBC	\$4600,X
11CD	F8			SED	
11CE	00			BRK	
11CF	FF			???	
11D0	F7			???	
11D1	00			BRK	
11D2	ED	F7	00	SBC	\$00F7
11D5	0D	20	20	ORA	\$2020
11D8	20	50	43	JSR	\$4350
11DB	20	20	53	JSR	\$5320
11DE	52			???	
11DF	20	41	43	JSR	\$4341
11E2	20	58	52	JSR	\$5258
11E5	20	59	52	JSR	\$5259
11E8	20	53	50	JSR	\$5053

## Extramon: A Machine Code Assembler

```
100 PRINT "TINY PEEKER/POKER"
110 X$="*": INPUT X$: IF X$="*" THEN END
120 GOSUB 500
130 IF E GOTO 280
140 A=V
150 IF J > LEN(X$) GOTO 300
160 FOR I=0 TO 7
170 P=J: GOSUB 550
180 C(I)=V
190 IF E GOTO 280
200 NEXT I
210 T=0
220 FOR I=0 TO 7
230 POKE A+I, C(I)
240 T=T+C(I)
250 NEXT I
260 PRINT "CHECKSUM="; T
270 GOTO 110
280 PRINT MID$(X$, 1, J); "??": GOTO 110
300 T=0
310 FOR I=0 TO 7
320 V=PEEK(A+I)
330 T=T+V
340 V=V/16
350 PRINT " ";
360 FOR J=1 TO 2
370 V%=V
380 V=(V-V%)*16
390 IF V% > 9 THEN V%=V%+7
400 PRINT CHR$(V%+48);
410 NEXT J
420 NEXT I
430 PRINT "/"; T
440 GOTO 110
500 P=1
510 L=4
520 GOTO 600
550 P=J
560 L=2
600 E=0
610 V=0
620 FOR J=P TO LEN(X$)
630 X=ASC(MID$(X$, J))
640 IF X=32 THEN NEXT J
650 IF J > LEN(X$) THEN 790
660 P=J
670 FOR J=PTOLEN(X$)
680 X=ASC(MID$(X$, J))
```

```

690 IF X<>32 THEN NEXT J
700 IF J-P<>L THEN 790
710 FORK=PTDJ-1
720 X=ASC(MID$(X$,K))
730 IF X<58 THENX=X-48
740 IF X>64 THENX=X-55
750 IF X<0 OR X>15 THEN790
760 V=V*16+X
770 NEXT K
780 RETURN
790 E=-1
800 RETURN

```

```

.10800 00 1A 0B 64 00 99 22 93 .10A00 02 20 4B FA 00 AD 3A 02 .10C00 60 A2 02 2C A2 00 00 84
.10808 12 1D 1D 1D 53 55 50 .10A08 20 4B FA 00 20 B7 FB 00 .10C08 C1 D0 0B 84 C2 D0 02 E6
.10810 45 52 20 36 34 2D 4D 4F .10A10 20 8D FB 00 F0 5C 20 3E .10C10 26 D6 C2 D6 C1 60 20 3E
.10818 4E 00 01 38 6E 00 99 22 .10A18 F8 00 20 79 FA 00 90 33 .10C18 FB 00 C9 20 F0 F9 60 99
.10820 11 20 20 20 20 20 20 20 .10A20 20 69 FA 00 20 3E FB 00 .10C20 00 00 8D 00 00 01 20 AC
.10828 20 20 20 20 20 20 20 20 .10A28 20 79 FA 00 90 2B 20 69 .10C28 FA 00 20 BF FA 00 20 7C
.10830 00 4B 0B 78 00 99 22 11 .10A30 FA 00 A9 90 2D D2 FF 20 .10C38 00 79 FA 00 80 DE AE
.10838 20 2E 2E 4A 49 40 2D 42 .10A38 E1 FF F0 3C A6 26 D0 3B .10C40 3F 02 9A A9 90 2D D2 FF
.10840 55 54 54 55 52 46 49 45 .10A40 A5 C3 C5 C1 A5 C4 E5 C2 .10C48 A9 3F 20 D2 FF 4C 47 FB
.10848 4C 44 00 66 0B 82 00 9E .10A48 90 2E A0 3A 20 C2 FB 00 .10C50 00 20 54 FD 00 CA D0 FA
.10850 2B C2 2B 34 33 29 AA 32 .10A50 20 41 FA 00 20 8B FB 00 .10C58 60 E6 C3 D0 02 E6 C4 60
.10858 35 36 AC C2 2B 34 34 29 .10A58 F0 E0 4C ED FA 00 20 79 .10C60 A2 02 B5 D0 4B 85 27 95
.10860 AA 31 32 37 29 00 00 00 .10A60 FA 00 90 03 20 80 FB 00 .10C68 C0 A8 95 27 CA D0 F3 60
.10868 AA AA AA AA AA AA AA AA .10A68 20 B7 FB 00 20 80 FB 00 .10C70 A5 C3 A4 C4 3B E9 02 B0
.10870 AA AA AA AA AA AA AA AA .10A70 FA 00 90 EB A9 0B 85 1D .10C78 0E 8B 90 0B A5 2B A4 29
.10878 AA AA AA AA AA AA AA AA .10A78 20 3E FB 00 20 A1 FB 00

.10880 A5 2D 85 22 A5 2E 85 23 .10AB0 D0 FB 4C 47 FB 00 20 CF .10CB0 4C 33 FB 00 A5 C3 A4 C4
.10888 A5 37 85 24 A5 38 85 25 .10AB8 FF C9 0D F0 0C C9 20 D0 .10CB8 3B E5 C1 B5 1E 9B E5 C2
.10890 A0 00 A5 22 D0 02 C6 23 .10AB0 D1 20 79 FA 00 90 03 20 .10CB0 A8 05 1E 60 2D 04 FA 00
.10898 C6 22 B1 22 D0 3C A5 22 .10AB8 00 FB 00 A9 90 2D D2 FF .10CB8 20 69 FA 00 20 E5 FA 00
.108A0 D0 02 C6 23 C6 22 B1 22 .10AB0 AE 3F 02 9A 7B A0 39 02 .10CB8 20 2F FB 00 20 69 FA 00
.108AB F0 21 85 26 A5 22 D0 02 .10AB8 4B AD 3A 02 4B AD 3B 02 .10CB0 90 15 A6 26 D0 64 20 28
.108AC C6 23 C6 22 B1 22 1B 65 .10AB0 4B AD 3C 02 AE 3D 02 AE .10CB8 FB 00 90 5F A1 C1 B1 C3
.108BB 24 A5 A5 26 65 25 4B A5 .10AB8 3E 02 40 A9 90 2D D2 FF .10CC0 20 05 FB 00 20 33 FB 00
.108C0 57 D0 02 26 3B 26 36 68 .10AC0 AE 3F 02 9A 6C 02 A0 A0 .10CC8 D0 EB 2C 2B FB 00 1B A5
.108C8 91 37 8A 4B A5 37 D0 02 .10AC8 01 84 84 84 89 8B 84 87 .10CD0 1E A5 C5 B5 9B 65 CA
.108D0 C6 38 C6 37 6B 91 37 1B .10AD0 84 90 B4 93 A9 40 85 BB .10CD8 85 C4 2D 0C FB 00 A6 26
.108D8 90 B6 C9 4F D0 ED A5 37 .10ADB A9 02 B5 BC 20 CF FF C9 .10CE0 D0 3D A1 C1 B1 C3 20 28
.108E0 85 33 A5 38 B5 34 AC 37 .10AE0 20 F0 F9 C9 00 F0 3B C9 .10CE8 FB 00 B0 34 20 BB FA 00
.108EB 00 4F 4F 4F 4F AE E6 FF .10AEB 22 D0 14 20 CF FF C9 22 .10CF0 20 BB FA 00 20 69 FA 00
.108F0 00 8D 16 03 AD E7 FF 00 .10AF0 F0 10 C9 00 F0 29 91 BB .10CF8 20 D4 FA 00 20 69 FA 00
.108FB BD 17 03 A9 80 20 90 FF .10AFB E6 B7 CB C0 10 D0 EC 4C

.10900 00 00 DB 68 BD 3E 02 6B .10B00 ED FA 00 20 CF FF C9 00 .10D00 20 E5 FA 00 20 69 FA 00
.10908 BD 3D 02 6B BD 3C 02 6B .10B08 F0 16 C9 2C D0 DC 20 8B .10D08 20 3E FB 00 20 BB FA 00
.10910 BD 3B 02 6B A6 68 AB 3B .10B10 FA 00 29 0F F0 E9 C9 03 .10D10 90 14 85 1D A6 26 D0 11
.10918 BA E9 02 8D 3A 02 9B E9 .10B18 F0 E5 B5 BA 20 CF FF C9 .10D18 20 2F FB 00 90 0C A5 1D
.10920 00 00 BD 39 02 BA BE 3F .10B20 D0 60 6C 30 03 AC 32 03 .10D20 B1 C1 20 33 FB 00 D0 EE
.10928 02 20 57 F0 00 A2 42 A9 .10B28 20 96 F9 00 D0 D4 A9 90 .10D28 AC ED FA 00 4C 47 FB 00
.10930 2A 20 57 FA 00 A9 52 D0 .10B30 20 D2 FF A9 00 20 EF .10D30 20 D4 FA 00 20 69 FA 00
.10938 34 E6 C1 D0 06 E6 C2 D0 .10B38 F9 00 A5 90 29 10 D0 C4 .10D38 20 E5 FA 00 20 69 FA 00
.10940 02 E6 26 60 20 CF FF C9 .10B40 AC 47 FB 00 20 96 F9 00 .10D40 20 3E FB 00 A2 00 00 20
.10948 D0 D0 FB 68 6B A9 90 20 .10B48 C9 2C D0 BA 20 79 FA 00 .10D48 3E FB 00 C9 27 D0 14 20
.10950 D2 FF A9 00 00 85 26 A2 .10B50 20 69 FA 00 20 CF FF C9 .10D50 3E FB 00 90 10 02 EB 20
.10958 D0 A9 2E 20 57 FA 00 A9 .10B58 2C D0 AD 20 79 FA 00 A5 .10D58 CF FF C9 00 F0 22 E0 20
.10960 05 20 D2 FF 20 3E FB 00 .10B60 C1 85 AE A5 C2 B5 AF 20 .10D60 D0 F1 F0 1C 8E 00 00 01
.10968 C9 2E F0 F9 C9 20 F0 F5 .10B68 69 FA 00 20 CF FF C9 00 .10D68 20 BF FA 00 20 69 FA 00
.10970 A2 0E D0 B7 FF 00 D0 0C .10B70 D0 9B A9 90 20 D2 FF 20 .10D70 02 EB 20 CF FF C9 00 B6 E0
.10978 BA 0A AD BD C7 FF 00 4B .10B78 F2 F9 00 4C 47 FB 00 A5 .10D78 09 20 BB FA 00 20 69 FA 00

.10980 BD C6 FF 00 4B 60 CA 10 .10B80 C2 20 4B FA 00 A5 C1 4B .10D80 20 D0 EC 86 1C A9 90 20
.10988 EC 4C ED FA 00 00 A5 C1 8D .10B88 4A 4A 4A 4A 20 60 FA 00 .10D88 D2 FF 20 57 FD 00 A2 00
.10990 3A 02 A5 C2 BD 39 02 60 .10B90 AA 6B 29 0F 20 60 FA 00 .10D90 00 A0 00 00 B1 C1 D0 1D
.10998 A9 0B 85 1D 00 00 00 20 .10B98 4B 8A 20 D2 FF 6B AC D2 .10D98 02 D0 0C EB E4 1C 10
.109A0 54 FD 00 B1 C1 20 4B FA .10BA0 FF 09 30 C9 3A 90 02 69 .10DA0 F3 20 41 FA 00 20 54 FD 00
.109AB 00 20 33 FB 00 CA 1D 00 .10BA8 06 60 A2 02 B5 C0 4B B5 .10DA8 00 20 33 FB 00 A6 26 D0
.109B0 F1 60 20 8B FA 00 90 0B .10BB0 C2 95 C0 6B 95 C2 CA D0 .10DB0 BD 20 2F FB 00 80 D0 4C
.109B8 A2 00 00 B1 C1 C1 C1 F0 .10BB8 F3 60 20 8B FA 00 90 02 .10DB8 47 FB 00 20 D4 FA 00 85
.109C0 03 4C ED FA 00 20 33 FB .10BC0 B5 C2 20 8B FA 00 90 02 .10DC0 20 A5 C2 B5 21 A2 00 00
.109CB 00 C6 1D 60 A9 3B 85 C1 .10BC8 B5 C1 60 A9 00 00 B5 2A .10DC8 86 2B A9 93 20 D2 FF A9
.109D0 A9 02 B5 C2 05 60 9B .10BD0 20 3E FB 00 C9 20 D0 0E .10DD0 90 20 D2 FF A9 16 85 1D
.109D8 4B 20 57 FD 00 A8 AD 2E .10BD8 20 3E FB 00 C9 20 D0 0E .10DD8 20 6A FC 00 20 CA FC 00
.109E0 AC 57 FA 00 A9 90 20 D2 .10BE0 1B 60 20 AF FA 00 0A 0A .10DE0 B5 C1 B4 C2 C6 1D D0 F2
.109EB FF A2 00 00 BD EA FF 00 .10BE8 0A 0A B5 2A 20 3E FB 00 .10DE8 A9 91 20 D2 FF 4C 47 FB
.109F0 20 D2 FF EB 0E 16 D0 F5 .10BF0 AF FA 00 05 2A 3B 60 .10DF0 A0 A0 2C 20 C2 FB 00 20
.109FB 80 3B 20 C2 FB 00 AD 39 .10BF8 C9 3A F0 02 69 0B 29 0F .10DF8 54 FD 20 20 41 FA 00 20

```

```

.10E00 54 FD 00 A2 00 00 A1 C1 .10F80 F6 CA D0 ED A2 02 20 CF .11100 59 00 00 58 24 24 00 00
.10E08 20 D9 FC 00 48 20 1F FD .10F88 FF C9 0D F0 1E C9 20 F0 .11108 1C 8A 1C 23 5D 8B 1B A1
.10E10 00 68 20 35 FD 00 A2 06 .10F90 F5 20 D0 FE 00 80 0F 20 .11110 9D 8A 1D 23 9D 8B 1D A1
.10E18 E0 03 D0 12 A4 1F F0 0E .10F98 9C FA 00 A4 C1 84 C2 85 .11118 00 00 29 19 AE 69 AB 19
.10E20 A5 2A C9 EB B1 C1 B0 1C .10FA0 C1 A9 30 90 10 02 EB 9D .11120 23 24 53 1B 23 24 53 19
.10E28 20 C2 FC 00 88 D0 F2 06 .10FAB 10 02 E8 D0 DB 86 2B A2 .11128 A1 00 00 1A 5B 5B A5 69
.10E30 2A 90 0E BD 2A FF 00 20 .10FB0 00 00 86 26 F0 04 E6 26 .11130 24 24 AE AE AB AD 29 00
.10E38 A5 FD 00 BD 30 FF 00 F0 .10FBB F0 75 A2 00 00 86 1D A5 .11138 00 7C 00 00 15 9C 6D 9C
.10E40 03 20 A5 FD 00 CA D0 D5 .10FC0 26 20 D9 FC 00 A6 2A 86 .11140 A5 69 29 53 84 13 34 11
.10E48 60 20 CD FC 00 AA EB D0 .10FC8 29 AA BC 37 FF 00 BD 77 .11148 A5 69 23 A0 DB 62 5A 4B
.10E50 01 CB 9B 20 C2 FC 00 0A .10FD0 FF 00 20 B9 FE 00 D0 E3 .11150 26 62 94 8B 54 44 CB 54
.10E58 B6 1C 20 48 FA 00 A6 1C .10FDB A2 06 E0 03 D0 19 A4 1F .11158 68 44 EB 94 00 00 B4 08
.10E60 60 A5 1F 3B A4 C2 AA 01 .10FE0 F0 15 A5 2A C9 EB A9 30 .11160 84 74 B4 2B 6E 74 F4 CC
.10E68 01 8B 65 C1 90 01 CB 60 .10FEB 00 21 20 BF FE 00 D0 CC .11168 4A 72 F2 A4 8A 00 00 AA
.10E70 AB 4A 90 0B 4A 80 17 C9 .10FF0 20 C1 FE 00 D0 C7 8B D0 .11170 A2 A2 74 74 74 72 44 6B
.10E78 22 F0 13 29 07 09 80 4A .10FFB EB 06 2A 90 0B BC 30 FF .11178 B2 32 B2 00 00 22 00 00

.10EB0 AA BD D9 FE 00 B0 04 4A .11000 00 BD 2A FF 00 20 B9 FE .11180 1A 1A 26 26 72 72 8B CB
.10EB8 AA 4A 4A 29 0F D0 04 A0 .11008 00 D0 B5 CA D0 D1 F0 0A .11188 C4 CA 26 4B 44 44 A2 CB
.10E90 80 A9 00 00 AA BD 1D FF .11010 20 8B FE 00 D0 AB 20 8B .11190 3A 3B 52 4D 47 5B AC 53
.10EB8 00 85 2A 29 03 85 1F 9B .11018 FE 00 D0 A6 A5 2B C5 1D .11198 54 46 4B 44 50 2C 41 42
.10EA0 29 BF AA 9B A0 03 E0 BA .11020 D0 A0 20 69 FA 00 A4 1F .111A0 F9 00 35 F9 00 CC FB 00
.10EAB F0 0B 4A 90 0B 4A 4A 09 .11028 F0 2B A5 29 C9 9D D0 1A .111A8 F7 F8 00 56 F9 00 89 F9
.10EB0 20 8B D0 FA CB 8B D0 F2 .11030 20 1C FB 00 90 0A 9B D0 .111B0 00 FA F9 00 0C FA 00 3E
.10E8B 60 B1 C1 20 C2 FC 00 A2 .11038 04 A5 1E 10 0A 4C ED FA .111B8 FB 00 92 FB 00 C0 FB 00
.10EC0 01 20 FE FA 00 C4 1F CB .11040 00 C8 D0 FA A5 1E 10 F6 .111C0 3B FC 00 5B FD 00 8A FD
.10EC8 90 F1 A2 03 C0 04 90 F2 .11048 A4 1F D0 03 89 C2 90 00 .111C8 00 AC FD 00 46 FB 00 FF
.10ED0 60 AB 99 37 FF 00 85 2B .11050 91 C1 8B D0 FB A5 26 91 .111D0 F7 00 ED F7 00 0D 20 20
.10ED8 B9 77 FF 00 85 29 A9 00 .11058 C1 20 CA FC 00 85 C1 84 .111D8 20 50 43 20 20 53 52 20
.10EE0 00 A0 05 06 29 26 2B 2A .11060 C2 A9 90 20 D2 FF A0 41 .111E0 A1 43 20 5B 52 20 59 52
.10EE8 B8 D0 FB 69 3F 20 D2 FF .11068 20 C2 FB 00 20 54 FD 00 .111E8 20 53 50 5B 80 01 22 20
.10EF0 CA D0 EC A9 20 2C A9 0D .11070 20 41 FA 00 20 54 FD 00
.10EF8 4C D2 FF 20 D4 FA 00 20 .11078 A9 05 20 D2 FF 4C 80 FD

.10F00 69 FA 00 20 E5 FA 00 20 .11080 00 AB 20 BF FE 00 D0 11
.10F08 69 FA 00 A2 00 00 86 2B .11088 9B F0 0E B6 1C A6 1D DD
.10F10 A9 90 20 D2 FF 20 57 FD .11090 10 02 0B EB 86 1D A6 1C
.10F18 00 20 72 FE 00 20 CA FC .11098 2B 60 C9 30 90 03 C9 47
.10F20 00 85 C1 84 C2 20 E1 FF .110A0 60 3B 60 40 02 45 03 D0
.10F28 F0 05 20 2F FB 00 80 E9 .110AB 0B 40 09 30 22 45 33 D0
.10F30 4C 47 FB 00 20 D4 FA 00 .110B0 0B 40 09 40 02 45 33 D0
.10F38 A9 03 85 1D 20 3E FB 00 .110BB 0B 40 09 40 02 45 83 D0
.10F40 20 A1 FB 00 D0 FB A5 20 .110C0 0B 40 09 00 00 22 44 33
.10F48 B5 C1 A5 21 B5 C2 4C 46 .110CB D0 BC 44 00 00 11 22 44
.10F50 FC 00 C5 2B F0 00 D2 .110D0 33 D0 BC 44 9A 10 22 44
.10F58 FF 60 20 D4 FA 00 20 69 .110DB 33 D0 0B 40 09 10 22 44
.10F60 FA 00 8E 11 02 A2 03 20 .110ED 33 D0 0B 40 09 62 13 7B
.10F68 CC FA 00 4B CA D0 F9 A2 .110EB A9 00 00 21 B1 82 00 00
.10F70 03 6B 3B E9 3F A0 05 4A .110FO 00 00 59 4D 91 92 86 A4
.10F78 6E 11 02 6F 10 02 8B D0 .110FB B5 9D 2C 29 2C 23 2B 24

```

# Instructions for entering and using Supermon64

## Entering the program

Enter in immediate mode:

POKE 8192,0:POKE 44,32:NEW <return>

This moves the start of Basic to decimal 12800, and gives us room to put in Supermon. Now type in Tiny Peeker/Poker.

Run the program, and in answer to the question prompt type in the memory address and memory contents as given in the 20 blocks of data at the end of this section. You can look at memory just by entering the memory address.

When you've finished, type in immediate mode :

POKE 44,0B:CLR <return>

which puts Basic back to normal again. You can now save Supermon by using a normal Basic SAVE. Before running it, you'll need to check that its all there, so in immediate mode, enter the following line (you'll need to use Basic abbreviations e.g. '?' for PRINT, P shifted E for PEEK, and so on, to fit it all in):

```
M=1:FORJ=0TO18:FORI=2048+128*NT02047+128*N:A=A+
PEEK(I):NEXTI:PRINTA;J:A=0:N=N+1:M=M+1:NEXTJ
```

This should display the following numbers on your screen, together with the block number :

```
10021, 13841, 14762, 15283, 14641, 16091, 16771,
13076, 15720, 14716, 14189, 15165, 14543, 15051,
14669, 16467, 16259, 9635, 11241
```

although they'll be in column form on the screen. Due to the vagaries of Basic the first number might differ (I got 12205 once!), but running the check again should sort it out. If one of your numbers disagrees, we'll need to go back to the beginning with Tiny Peeker again. So, enter the first set of POKEs again (Supermon will still be there), enter or re-load Tiny Peeker, and check each incorrect block. The last block will have to be checked by hand.

When you've done that, enter POKE 44,08:CLR again, re-SAVE Supermon, and run the checking program. When you've finally got it right, the 64 is yours for the disassembling!

#### Using Supermon 64

This will be given in the form COMMAND, followed by the syntax.

##### 1) Simple Assembler

```
.A 2000 LDA#12
start assembly at 2000 hex.
```

##### 2) Disassembler

```
.D 2000
disassemble hex from 2000 onwards.
```

##### 3) Printing Disassembler

```
.P 2000,2040
engage printer beforehand with OPEN4,4:CMD4.
```

##### 4) Fill memory

```
.F 1000 1100 FF
fill memory from 1000 to 1100 hex with the byte FF.
```

##### 5) Go run

```
.G 1000
go to hex 1000 and execute program there.
```

##### 6) Hunt memory

```
.H C000 D000 'READ
look from C000 to D000 for the ASCII string READ.
```

##### 7) Load

```
.L "FRED",08
```

##### 8) Memory display

```
.H 0800 0820
display memory from hex 0800 to 0820.
```

##### 9) Register display

```
.R
displays register values when Extramon was entered.
```

##### 10) Save

```
.S "OFRED",08,0800,0820
save memory from hex 0800 to 0820 onto device 08 drive 1, and call that portion of memory FRED.
```

##### 11) Transfer memory

```
.T 1000 1100 5000
transfer memory in the range hex 1000 to 1100 and start storing it at hex 5000 onwards.
```

##### 12) Exit to Basic

```
.D
return to Basic ready mode. Perform a CLR before doing anything.
```

# Index

6566 Video Interface Chip, 78-94  
6581 video chip description, 170-8  
Adding commands to Basic, 203  
Adding commands: concepts, 209; characters, 216, 217  
Advanced keyboard listing, 193-6  
Alpine Slopes listing, 67-70  
Android Nim, 34, 35; listing, 36-41  
ASCII characters, 31, 32  
Assembler listing, 224-9  
Assembler: Basic program, 250-1; hex dump, 251, 252;  
    commands, 252, 253  
Attack/decay/sustain/release/listing, 184  
Attack/decay settings, 154  
Background colours, 21  
Bit map code, 82-4  
Bits, 12, 13, 14  
Border colours, 21  
Bouncing blob listing, 20, 21  
Bytes, 12, 13, 14  
Character display mode, 78-80  
Character memory map, 97, 98  
Character selection, 98  
Character set: altering, 100-2  
Character generator listing, 112-15  
Character Get routine, 210, 211; alteration, 214, 215  
CHR\$ characters, 28, 29, 30  
Colour selection, 27, 28  
Colours available, 19  
Commodore 64 Basic, 206, 208  
Commodore Basic, 205  
CTRL key, 20  
Cursor codes, 22  
Deathtrap listing, 198-201  
Decimal, 12, 13, 14  
Dec/binary convertor, 204  
Duckworth duck listing, 51  
Envelope generating, 147, 149, 150, 179  
Envelope rates, 175  
Envelope shapes, 180  
Extended colour mode, 81, 82, 109  
Filtering, 148, 149, 176-8, 185, 186

- Function keys, 26, 219-22
- Graph plotting, 33, 34
- Graphics modes, 105, 106
- Hexadecimal, 12, 13, 14
- Hex/dec convertor, 204
- High Resolution introduction, 121
- High resolution m/c routines, 134-7
- Interrupt registers, 90, 91
- Keyboard tour, 25, 26
- Keyboard tricks, 33
- Light pens, 90
- Lower case, 30
- Memory interfacing, 93, 94
- Memory banking, 99
- Memory architecture, 207
- Movemaze listing, 116-19
- Multiple voice manipulation, 168, 169
- Multi-colour char. mode, 80, 81, 106
- Multi-colour bit mapping, 126
- Music: some physics, 162, 163
- Musical symbols, definitions, 140-2
- Musical notes table, 156, 157
- Musical tunes, 158, 159
- Musical values, 161, 162
- Musical keyboard listing, 165-7
- Musical instruments, 189-90
- OLD command listing, 217
- OR command, 14
- Pulse widths, 189
- Raster register, 90
- Ring modulation, 148, 172, 187-8
- Screen resolution, 15
- Screen locations, 18
- Screen colours, 19
- Screen blanking, 88
- Screen scrolling, 89, 90
- Screen: row & column select, 89
- SID chip overview, 144, 145
- SID registers, 145, 146
- SID memory map, 152, 153
- Sound: an introduction, 139-44
- Space Battle listing, 56-9
- Sprite resolution, 43
- Sprite memory, 44, 50, 87, 88
- Sprite data map, 46, 47
- Sprite generator listing, 61-5
- Sprites: an introduction, 43;
  - multi-coloured, 44, 45, 52; defining them, 48, 49, 85; positioning them, 52, 53, 84; priority, 53, 54, 86; turning off, 54, 84; multi-colour listing, 72-7
- Standard character mode, 80
- Standard bit mapping, 122-6
- Sustain/release settings, 155
- Synchronisation, 148, 172, 187-8
- The Thinker listing, 130-4
- User-defined graphics: introduction, 95, 96; defining them, 96, 97; re-defining keys, 97-103; storing them, 103
- Upper case, 30
- Voice 1, 170-5; filtering, 170; pulse width, 171;
  - control register, 171-3
- Voices 2 and 3, 176
- Waveform selection, 155
- Waveforms, 181-2





## **DUCKWORTH HOME COMPUTING**

All books written by Peter Gerrard, former editor of *Commodore Computing International*, author of two top-selling adventure games for the Commodore 64, or by Kevin Bergin. Both are regular contributors to *Personal Computer News*, *Which Micro?* and *Software Review*.

### **THE COMPLETE 64 ROM DISASSEMBLY**

by Peter Gerrard and Kevin Bergin

This book is for anyone who has ever wondered how the Commodore 64 really works. Intended for the serious programmer, it includes fundamental memory maps, memory architecture maps, the disassembly itself and (for reference) the complete 6510 machine code instruction set. £5.95

### **ADVANCED BASIC & MACHINE CODE FOR THE 64**

by Peter Gerrard

For the more serious user of the Commodore 64, this book teaches you all about programming in Machine Code, with sections on double precision arithmetic and animation, along with a series of chapters on using the special features of the Commodore 64. The all-important link to Basic is not forgotten, and the opening chapters form a guide to improving your Basic programming techniques, along with many program examples. £6.95

Other titles in the series include *Using the 64*, *12 Simple Electronic Projects for the VIC*, *Will You Still Love Me When I'm 64*, *Advanced Basic & Machine Code Programming on the VIC*, as well as *Pocket Handbooks for the VIC*, *64*, *Dragon*, *Spectrum* and *BBC Model B*.

*Write in for a descriptive leaflet (with details of cassettes).*



**DUCKWORTH**

The Old Piano Factory, 43 Gloucester Crescent, London NW1 7DY  
Tel: 01-485 3484

# Duckworth Home Computing

## **SPRITES & SOUND ON THE COMMODORE 64**

**by Peter Gerrard**

This is a complete guide to the best features of the 64. There are chapters on sprites, user-defined graphics, hi-res plotting, advanced sound programming techniques such as ring modulation and filtering, adding commands to Basic and a disassembly and explanation of Extramon. There is also a full explanation of the chips responsible for the 64's graphics and sound: the 6566 Video Interface Chip and the 6581 Sound Interface Device. Sections on how to program your own musical instruments and how to produce sprite and programmable character animation make this the essential guide for anyone who wants to get the most from the 64's special features.

Peter Gerrard, former editor of *Commodore Computing International*, is the author of two top-selling adventure games for the Commodore 64 and a regular contributor to *Personal Computer News*, *Which Micro?* and *Software Review* and *Commodore Horizons*.

ISBN 0-7156-1781-8



9 780715 617816

**Duckworth**

The Old Piano Factory  
43 Gloucester Crescent, London NW1

ISBN 0 7156 1781 8

IN UK ONLY £6.95 NET