Pan

PERSONAL COMPUTER

COMPUTER NEWS LIBRARY

KEITH BOWDEN

The COMPANION to the COMMODORE 64

Pan/Personal Computer News
Computer Library

Keith Bowden

The Companion to the Commodore 64

Pan Books London and Sydney

First published 1984 by Pan Books Ltd,
Cavaye Place, London SW10 9PG
in association with Personal Computer News
9 8 7 6 5 4 3 2 1

ISBN 0 330 28479 7
Photoset by Parker Typesetting Service, Leicester
Printed and bound in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk

To **Valerie Buckle**,

A Rasta Interrupt

Acknowledgements

This book would have been impossible without the help of a great number of people. I am indebted to Steve Beats at Commodore for technical assistance whilst I was writing the programs, and to Gail Wellington for supplying copies of Easy Script, on which the text was mostly written and Petspeed, which greatly speeded up the testing of the BASIC programs (it is well worthwhile compiling them). Thanks are also due to many of the staff at Pan/PCN, especially Richard Kennedy for initiating the project and seeing it through some rough spots, and to Keith Elliot for permission to use those reviews which previously appeared in *What Micro*. I am also grateful to the many friends that have given me their support, to Trevor Elliott and Valerie Preston for typing the greater part of the manuscript, to Clive Emberey who came to the rescue when I felt that I could not continue running the user club and to Valerie Buckle for doing all the hard work.

Contents

Introduction

This book is aimed at users of the Commodore 64 microcomputer who wish to make more creative use of one of the most powerful home computers on the market. It is intended to represent the state-of-the-art in programming sound and graphics on the 64 although, as its title suggests, the text can also be regarded as a companion to the *Commodore 64 Reference Guide*, which is also helpful in making full use of your computer. The latter is one of the best pieces of computer documentation I have seen. On the other hand, although it contains most of the information necessary to make full use of the machine, there is little on how to use (or make creative use of), this knowledge. While you are making your way through the text, it would be useful, although not essential, to work with a graphics package such as Abacus Software's Screen Graphics 64 and an Assembler Monitor such as Supermon (which is available free via the user clubs and was published in the January 1983 issue of *Compute*). All programs in this book will work with either cassette or disc-based systems.

It is not our intention to teach the reader BASIC or machine code, although we do introduce the elements of 6502 assembler and show how simple yet very powerful routines can be effectively exploited. These are reproduced as BASIC POKE routines for those without an assembler. The approach of this book is close to that of Commodore when designing the machine. We explain the workings of the computer in terms of PEEKs and POKEs. The potential games programming genius will quickly be able to translate these ideas into machine code. The user of Simons' BASIC will learn what actually happens when he calls a routine to move a sprite. Finally the middle of the road user should find that he learns all the tricks and ideas that can help him get the most out of his machine.

So just how does the Commodore 64 compare with its rivals? It has the most powerful synthesiser chip (known as SID) available on any home micro. SID has three voices (three notes polyphonic) and each voice is separately synthesised and controlled and can have three basic waveforms or be random noise. The envelope shape of each can be controlled in the same way as the Casio VL-Tone (ADSR). Each note can then be filtered with programmable filters. These can be changed dynamically while the music is playing to produce phasing and waawaa effects, etc. Synchronisation (e.g. tremelo and vibrato), and ring modulation (as used by Hendrix) are also available. The sound quality is superb.

There are two graphics modes (excluding sprite graphics and character graphics) which can be mixed with text by splitting the screen:

(1) High resolution bit mapped graphics uses 320×200 pixels with two colours out of sixteen in each 8×8 pixel character square (as on the Sinclair Spectrum). This sixteen-colour high-resolution mode fits into any 9K RAM (compare 20K with sixteen colours and a lower resolution, for true high resolution graphics, in the equivalent mode on the BBC micro) and thus more than one page of graphics can be stored.

(2) Multicolour mode uses 160×200 pixels with four colours out of sixteen in each character square. We still have sixteen colours in 9K RAM but this mode overcomes most of the limitations of the high-resolution mode.

The sixteen colours can be increased to over fifty by colour mixing. My only complaint about the 64's graphics is that the colour resolution is not too good. However there is no dot crawl, no flicker and there are no rolling bands of colour.

The sprites are amazing (much better than Atari's). Each one can be in a different colour mode. Not only that but each one is not just an area of memory, but a pointer to an area of memory. You can move the pointer using just one POKE, so that the sprites don't just move around but can be fully animated, in colour, by moving the pointer between different 64-byte blocks of memory, each containing a frame of the animation – and all from BASIC.

There are four layers of animation and eight sprite layers. High resolution and multicolour mode graphics and the three text modes allow parts of the screen to be defined as foreground and parts to be defined as background so that sprites can pass in front of or behind the foreground. Collisions can be detected between sprites, sprite and foreground, or sprite and border.

There is a raster interrupt register in the Commodore 64 to tell you how far down the screen the television raster has reached. This enables split screen work to be carried out and allows the number of sprites on the screen to be increased from eight to seventy-two (not 256 as claimed by Commodore. 256 is actually the maximum number of sprite data frames than can be seen by the VIC chip at once). Sprite collisions also cause interrupts.

Completing your system

Commodore 64 BASIC is standard Microsoft Pet BASIC V2, with no graphics or sound commands, but including the Pet graphics and colour control symbols. It leaves 39K of BASIC RAM available to the user. So if the 64 lacks a sophisticated BASIC what packages should you buy to fill in the gaps? Simons' BASIC is available as a cartridge or disc and includes most of the commands (114 of them) one would have expected in the original system,

including structured BASIC, graphics, sprites, sound, toolkit (no DELETE), screen save and restore and disc commands. Simons' BASIC II is a soft loaded extension to SB and includes matrix algebra, travelling sprites, copy between sprite and screen – and, finally, DELETE. Simons' BASIC leaves 31K BASIC user RAM and SBII leaves 27K. Personally, I am not keen on Simons' BASIC, except for beginners. It is expensive, not implemented as well as it could have been, and obscures the workings of the machine from the user. Better to struggle with PEEKs and POKEs in BASIC (this is undoubtedly the best road to learning machine language) and buy a graphics package such as Screen Graphics 64 (which leaves 39K of user RAM free) for when you want to plot lines and circles. Full reviews of many of the available packages are given in Appendix B. Simons BASIC is, of course, helpful in writing structured programs but it is not strictly necessary; a structured approach can be used with advantage at all levels of programming languages from ADA to machine code.

What extra hardware do you need to complete your computer set-up? Well, you can get by quite happily with just a cassette recorder (I did for six months), but disc drive and printer are a boon if you can afford them. The new Commodore C2N cassette recorder is highly recommended. Other firms have produced adaptors for ordinary cassette recorders but these have proved unreliable. Don't skimp on this all-important investment.

The VIC 1541 disc drive is an intelligent unit containing a 6502 microprocessor and 2K RAM. It is capable of performing tasks, such as formatting a disc, without tying up your computer. A number of disc drives and a printer can be daisy chained together and the utility disc provided with the drive includes a (very slow) routine for backing up discs on two drives. Hardware reliability is high but problems have been known to occur when a disc becomes full. Keep backups on cassette! Unfortunately the Commodore 64 DOS is still based on the old Pet BASIC V2. To delete a file you must type:

```
OPEN 15,8,15,"S0:filename":CLOSE 15
```

The discs will store 170K each, formatted.

The new 1525 printer is a ten-inch-wide, tractor-fed, dot-matrix printer. It contains the full Pet graphical character set in ROM, will also print user defined graphics, and, using a package such as UltraBasic, allows dumping of the high-resolution graphics screen. It is rather noisy in use, does not have true descenders, and my own machine had teething problems, but the older 1515 has received good reports. Commodore also sells a very nice colour graphics plotter and an excellent colour monitor.

In the UK Commodore 64 users can access both Micronet 800 and Prestel via either the Micronet telephone modem or Commodore's own direct connection network adaptor, for around the same price. The latter also allows users to access Commodore's Petnet. Micronet 800 allows users to down-

load at least a hundred free programs (telesoftware), sell, buy or try programs via the network, send messages, run magazines or hold conferences. In America there are even 'discussion trees' growing on these networks.

What's in this book

In this book we try to teach by example, while at the same time giving as detailed a tour of the sound and graphics facilities in the Commodore 64 – and how to use them – as possible. To achieve this we present three major BASIC programs, a synthesiser package (the most sophisticated of its kind available) to allow you to experiment with all the facilities of the SID chip and create your own sounds, two programs to allow you to create detailed bit mapped pictures on the screen and save them on cassette or disc, one program for high-resolution mode and one for multicolour mode. The important thing about these three programs is the set of subroutines which go with each of them. These are exactly the set of routines you need to complement Commodore BASIC and allow you to use your microcomputer to the full. Routines are given for switching to graphics, back to text, plotting a point, filling an area, etc. and for driving the synthesiser chip. Each one is explained in detail, POKE by POKE, or even bit by bit where necessary! Some of the graphics routines are rather slow and must be translated to machine language, or compiled, if speed is a consideration, but they are quite adequate for use with the programs given – and give a detailed grounding in the fundamentals of programming the Commodore 64's remarkable hardware.

The second major direction that this book takes is towards the use of machine code interrupt routines as the only sensible, and most powerful, way of driving animated displays and/or music generation. BASIC versions of all routines are included. These can be used simply as utilities or extended by the user; or even integrated into his own programs. Programming real time applications such as animation or music necessitates a great deal of interleaving of routines to move objects around the screen, calculate where they must go next, find whether there have been any collisions between objects or inputs to the program, decide what is the next note to play and generally run the organisation of the program. The sprites must move smoothly and the notes play at the correct time. This synchronisation could be achieved for relatively simple applications by the BASIC or machine language programmer, given a great deal of effort. The problem is that home computers can usually only do one thing at once. A simulation of multiprocessing, doing more from one thing at once, can be achieved by making use of interrupt routines. In Chapter 3 an efficient music interpreter is presented which can run at the same time as BASIC. One can develop or run a completely different program and still have the theme tune from the *Magic Roundabout* playing in the background, with only a slight decrease in the

speed of the machine! Alternatively a second program can be run which passes messages to the music interpreter, telling it to change voices or tempo etc. The spacing between notes is synchronised automatically by the system clock.

In the last chapter of the book we look at raster interrupts, which allow split screens to be created, or screen animation to be synchronised with the television raster scan, thus eliminating screen flicker and allowing easy interleaving of tasks. These also provide a means of increasing the number of sprites available to seventy-two. A remarkable system of 'False Video Chips' is introduced whereby the user apparently has access to an unlimited number of 'virtual' VIC chips which can be used simultaneously. Sprites can be made to travel across the screen automatically, with only the direction of movement being decided by your BASIC program, and animated or rotating sprites can be created. It is as if the architecture of the machine can be changed to your own design! All the programs are given as BASIC POKE routines with full instructions for use, and as assembler listings with line by line explanations of how they work.

Finally a quick chapter by chapter explanation of the structure of this book. Chapter 1 covers machine architecture and 6510 memory organisation, introduces the necessary elements of machine code and discusses interrupt routines. We then show how to translate BASIC programs into assembly language and assembly language into hexadecimal, and how to save machine code programs as DATA statements. Chapters 3 and 4 cover the SID chip, sound and music with a full description of envelope shaping, timbre and filtering, phasing, dynamic filtering (wah-wah), ring modulation, synchronisation, vibrato and tremelo, white and pink noise. We give an explanation of music for programmers, briefly cover scales, keys and chords, time signatures, bars and note lengths and finish with interrupt driven music generation with a full explanation of how the program works and how to use it to create your own background music. Chapter 5 introduces graphics and the VIC chips, and covers graphics modes, making the most of colour graphics, colour mixing, and saving and restoring the screen. Chapters 5 and 6 cover high resolution and multicolour bit mapped modes respectively, with detailed explanations of the routines mentioned above. Chapter 7 covers character and sprite animation, layers and sprite rotation and Chapter 8 is a full treatment of raster interrupts.

1 Machine Language

This chapter is not intended to be an introduction to machine code. However, it does attempt to cover all the material necessary to implement and understand the techniques used in the book, with particular reference to interrupt routines. Beginners may skip this chapter on first reading of the book and go on to implement the BASIC programs first.

Machine Architecture

The Commodore 64 is a MOS 6510 processor based microcomputer with 64K of RAM overlaid by 20K of ROM containing the BASIC interpreter, Kernel operating system and 4K of character ROM. The 6510 has the same machine language operator set and assembly language as the well tried 6502 chip but more sophisticated memory management. It allows complete reorganisation of the memory addressing structure of the machine. Any 8K block of the addressing range can be overlaid with layers of ROM or RAM and switched in or out by software when necessary. The 64 contains 64K of RAM covering the whole of the 6510's addressing range. This means that there are corresponding RAM locations which 'share' the same addresses with the ROM chips. For example, to copy the BASIC ROM into RAM we use PEEK(A) to read the ROM value at location A. POKE A,B will put into RAM location A the value B! Thus POKE A, PEEK(A) will perform the required copy. If it is desired to load a machine language program into memory without reducing the area available to BASIC, then the loader can be written in BASIC and the machine code program loaded under the BASIC interpreter itself. The BASIC interpreter in the ROM will execute the instructions in the BASIC loader program and POKE the machine code values into the RAM underneath itself, even though they both have the same addresses.

The Commodore 64 also includes two purpose designed chips: SID, or Sound Interface Device, and VIC II, the successor to the Video Interface Chip in Commodore's best selling VIC 20. SID interfaces with twenty-nine bytes of the normal 64K RAM. By POKEing values into these control registers the various tone generators, filters and modulators in the music synthesiser can be switched on or off or modified, and then left to get on with their respective jobs while another process runs in the 6510. By PEEKing other registers the state of control port peripherals can be read. Similarly the VIC

chip, 'sees' forty-seven bytes of RAM registers, which control sprites, graphics modes, etc. However, VIC can also 'see' any 16K block of memory in the 64's RAM. 8K of this can be used for high resolution bit mapped graphics. The rest contains sprite and colour data and the BASIC text screen.

Hexadecimal Notation

As computers work with binary (base 2) numbers, 1K (kilobyte) is actually equal to 2 ↑ 10=1024 bytes, where a byte is the smallest unit of memory that can be addressed, i.e. changed by a POKE or read using a PEEK. Each byte contains eight bits or binary digits which may hold either a '0' or a '1'. A byte can represent a number between 0 (every bit set to 0) and 255 (every bit set to 1) and there are thus 256 (=2 ↑ 8) possibilities. The lowest (rightmost) bit of the byte is referred to as bit 0, and the highest (leftmost) bit as bit 7. You should note that BASIC integer values use two bytes to store numbers −32768 to +32767, though we often refer only to the bottom eight bits (low-byte) when talking about POKEing or PEEKing memory.

When working in machine language, decimal notation is rather clumsy. It is not immediately obvious that if we want, say, a byte with the top four bits set (1) and the bottom four bits unset (0) that we need to POKE with the binary number 11110000=2 ↑ 7+2 ↑ 6+2 ↑ 5+2 ↑ 4=240 decimal.

Binary is equally clumsy as eight digits must be specified for each byte, and therefore programmers use hexadecimal notation. Each byte is split into two halves each of four bits, known as a nybble. Each nybble can hold a number between 0 and 15 (2 ↑ 4=16 possibilities), and its contents are represented by a digit from the set 0 to 9, plus the letters A to F, where A=10, B=11, C=12, D=13, E=14 and F=15. So our example byte would be written as F0 hexadecimal=15*16+0=240 decimal, which is an easier conversion.

Each memory location byte in the computer can be addressed by a number in the range $0000 to $FFFF hexadecimal, where $FFFF= (15*16 ↑ 3+15*16 ↑ 2+15*16 ↑ 1+15*16 ↑ 0)=64*1024−1=65535 or 64K possibilities. Each two-byte address requires four nybbles or hexadecimal digits to specify it. The '$' means that the number is in hexadecimal. Decimal/hexadecimal conversion tables are given in Appendix C.

To complicate matters further, if the contents of a particular byte represents either a variable, or a relative address (for which see below) and its value is greater than 127 decimal ($7F in hex) then it is taken to be negative. Thus we can store 256 numbers in the range −128 to +127:

Hex value:	$00	$01	. . .	$7F	$80	$81	. . .	$FF
Decimal value:	0	1	. . .	127	−128	−127	. . .	−1

The top or leftmost bit (bit 7) of the byte functions as a sign bit to show that the number is negative and to get the actual value of a negative byte you

must subtract 256 decimal. For example: $81−256=(129−256)=−127. Similarly, without a program in memory PRINT FRE(0) will give a value of −26627. To get the correct value we add 65536 (64*1024), giving (65536−26627)=38909 bytes of free memory.

Registers and the Stack

There are five 8-bit registers and two 16-bit registers used by the 6510 processor. The first three are known as the accumulator (or **A**-register) and the index, or **X**- and **Y**- registers. Each performs a slightly different function, but essentially they are equivalent to variables in BASIC. The accumulator is the most important as this is the sole register that allows arithmetic to be carried out (and then only addition and subtraction directly). The index registers are important for specifying relative addresses, i.e. those offset a certain number of bytes from a specified memory location, and in making comparisons. The other four registers are: the Processor status register, the bits of which are used as flags to indicate the state of the system; the 16-bit Program Counter, which contains the address of the machine code statement currently executing; the Stack pointer; and the input/output port (the other 16 bit register) which appears at locations 0 and 1 and is used to specify which bank of RAM or ROM should be in a particular area at a particular time.

The stack is an area of memory used to temporarily store values taken from the **A**, **X** and **Y** registers, subroutine return addresses and processor status whilst executing a subroutine or interrupt routine. It works on the 'Last In – First Out' principle so that the values can be immediately restored on return from the routine. The Stack pointer tells the processor the location in memory of the top of the stack. This is updated automatically on performing a stack operation.

BASIC to Assembler

As a 6510 machine code program simply consists of a set of hexadecimal numbers stored in the computer, a set of mnemonics known as assembly language has been devised to let you write programs more easily. You should try and obtain a monitor/assembler such as Supermon to allow you to input and list machine language programs in this form.

The first instructions we will look at here are LDA, which places (loads) a value into the accumulator or **A**-register, and STA, which stores the contents of the accumulator in a specified location. These can be used in an equivalent way to the POKE instruction in BASIC. Thus:

```
LDA #$08
STA $BCDE
```

will load the accumulator with the value $08 and store it in location $BCDE. This is known as immediate mode addressing, and is indicated by a hash sign, #. Similarly:

```
LDX #$08
STX $BCDE
```

achieves the same effect using the equivalent **X** register instructions. The sequence:

```
LDA $89AB
STA $BCDE
```

would copy the contents of location $89AB and store this value, via the accumulator, in $BCDE. Note that you are restricted to one machine code instruction per line. As for arithmetic:

```
ADC #$BC
```

will add (with carry) the value $BC to the value stored in the accumulator. If there is a carry (result over 255) then the Carry bit of the Status register is set. The instructions INX, INY, DEX, DEY increment (add 1) or decrement (take away 1) the **X** and **Y** register values. There are no equivalent instructions for the accumulator. The instruction equivalent to the GOTO in BASIC instruction is JMP (jump), which takes us to the specified location in memory, e.g. JMP $BCDE passes control to location $BCDE. Similarly, JSR is equivalent to a GOSUB statement and RTS to RETURN. To perform a conditional test we need two new statements. The sequence:

```
CPX #$55
BEQ $BCDE
```

instructs the machine to first compare the contents of the **X** register with the value stored at location $55 and then branch, if the two are equal, to location $BCDE. Note that this location must be within 128 bytes (up or down) of the location at which the CPX command is stored, since it is stored as a one-byte relative address. The assembler will convert the two-byte address into a single byte relative address. We have now seen enough commands to write a simple loop:

```
      LDX #$03
BCDE: NOP
      . . .
      INX
      CPX #$08
      BNE $BCDE
```

This is equivalent to: FOR X=3 TO 8 perform the section indicated by '. . .' The NOP command means 'no operation' (i.e. continue) and is stored at

location $BCDE. BNE means 'branch if not equal'. To stop a machine language program the equivalent of the STOP statement in BASIC is BRK (break). To return to BASIC however, after a call to a machine language subroutine via the SYS statement, the subroutine must end with RTS.

Addressing Modes

We have seen how to load a value into the accumulator in both immediate mode (LDA #$FE), and direct mode (LDA $ABCD). There are a number of other addressing modes applicable to some or all of the instructions and their associated registers.

LDA $ABCD, X

means load **A** from the address $ABCD as indexed by the contents of the **X** register. That is, if the **X** register contains $02, then the **A** register would be loaded with the contents of $ABCD+$02=$ABCF. The **A** register can be indexed by both **X** and **Y** registers, but the **X** and **Y** registers may only be indexed by each other. This is relative, or indexed addressing.

LDA $AB

will have the effect of loading the **A** register with the contents of location $00AB. This is known as Zero-page addressing, which allows us to have three new modes. (Zero-page locations are $00 to $FF.)

LDA $AB, Y
LDA $AB, X

This zero page indexed mode is fairly obvious. The indirect indexed mode instruction:

LDA ($AB), Y

Causes the **A** register to be loaded with the contents of (the location whose address is stored in bytes $AB *and* $AC) indexed by the **Y** register value. Note this is *indirect indexed* mode. *Indexed indirect* instructions have the form:

LDA ($AB, X)

This will load the **A** register with the contents of the location whose address is stored in the two locations starting at ($AB indexed by **X**). Note that 6500 series processors store the low byte of an address first. Hence, for example, the high byte of the address required for the indirect indexed example given above is stored in $AC and the low byte in $AB, so that, if **Y** contains $02, $00AB contains $89 and $00AC contains $67, then

LDA ($AB), Y

will form a two byte address from the contents of $00AC (high byte) and

$00AB, (low byte) giving $6789, and then index by the **Y** register value, giving $6789 + $02. Finally it will load the contents of the resulting address ($678B) into the **A** register.

Indexed indirect addressing is sometimes conveniently used for adding a series of numbers to the contents of a particular location. Indirect indexed may be used for obtaining the contents of a series of locations. Examples of such usage will be seen in the forthcoming chapters.

Bit Masking

This subject causes as much confusion in BASIC as it does in machine code. We will use BASIC notation to explain it. Equivalent machine code instruction examples are:

```
AND #$47
```

which ANDs the contents of the **A** register (accumulator) with $47,

```
ORA #$56
```

which ORs the contents of the accumulator with $56, and

```
EOR $BCDE
```

which will perform an Exclusive OR operation of the value stored in $BCDE with the accumulator value. To make full use of the SID and VIC chips in the 64 it is often necessary to change or test both single bits and nybbles in a particular memory location. This must be carried out using an extension of Boolean algebra. Remember that according to the truth tables the result of the AND operation is '1' (true) only if both of its parameters are '1' and the result of an OR operation is true if either or both of its parameters are '1'.

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

6510 machine code extends this to 8-bit binary values, thus, if X=10101010 (170 decimal) and Y=11110000 (240 decimal) we have:

```
    X=10101010          X=10101010
    Y=11110000          Y=11110000
X AND Y=10100000    X OR Y=11111010
```

In normal BASIC, using decimal values, these logical bitwise operations give:

```
170 AND 240=160
170 OR  240=246
```

NOT 240 will give 15. This is produced by 'flipping' all the 0's and 1's in a binary value:

```
      Y=11110000
NOT Y=00001111
```

Using these operations we can change any bits in a byte that we care to choose. For instance, if we need to invert bits 2 and 3 only of a variable VL we would use the following formula:

```
VL=NOT(VL AND(2 ↑ 2+2 ↑ 3)) OR (VL AND (255-(2 ↑ 2+2 ↑ 3)))
```

The statement VL AND (2 ↑ 2+2 ↑ 3) picks out bits 2 and 3 of VL, while VL AND (255-(2 ↑ 2+2 ↑ 3)) picks out all the others. The first bracket is inverted by the NOT operator, thus inverting bits 2 and 3, and then the OR puts everything back together again.

Interrupt Routines

Most of the machine language programs in this book are short interrupt routines. The 64's clock sends a message every 1/50th second to the 6510 processor to perform a set of service routines, such as scanning the keyboard for input and flashing the cursor. It is possible to divert this process to perform your own interrupt-driven routines before returning control to BASIC (or a standard machine language program) via the inbuilt service routines. This has the advantage that the interrupt routine runs as a background job and synchronisation of events is greatly simplified.

The normal interrupt entry point is given by the address contained in $FFFF and $FFFE, which is $FF48. The routine at this location saves the **X**, **Y** and **A** registers and jumps to the address stored at $0315 and $0314, which in turn is usually location $EA31. So all we need to do is change the address stored at $0314/5 to point to our own routine and then finish by jumping back to $EA31. The only restriction is that we must switch off interrupts using the SEI command before we change this address, and then switch them on again afterwards using CLI. Otherwise we risk an interrupt occurring half way through a double byte address change which could cause a jump to entirely the wrong location. If it is desired to miss out the normal interrupt service routines then the jump back can be made to $EA81 (instead of $EA31) which simply restores the **A**,**X** and **Y** registers. By alternating jumps to $EA31 and $EA81 the cursor and keyboard input are slowed down by half! Should the user wish to have total control over what goes on he can switch in a RAM area at the end of memory and change the address at $FFFE/F to point to his own

interrupt routine, simply ending the routine with an RTI (return from interrupt) instruction. All registers and input/output must then be handled by the user, and this is complex and not recommended.

If your interrupt routine is at $C000 (a good place) it might look like this:

```
C000:   interrupt routine goes here
        . . .
C0FF:   JMP $EA31
C100:   SEI
        LDA #$00
        STA $0314
        LDA #$C0
        STA $0315
        CLI
        BRK
        . . .
C200:   SEI
        LDA #$31
        STA $0314
        LDA #$EA
        STA $0315
        CLI
        BRK
```

To start the interrupt routine going type:

```
G C100
```

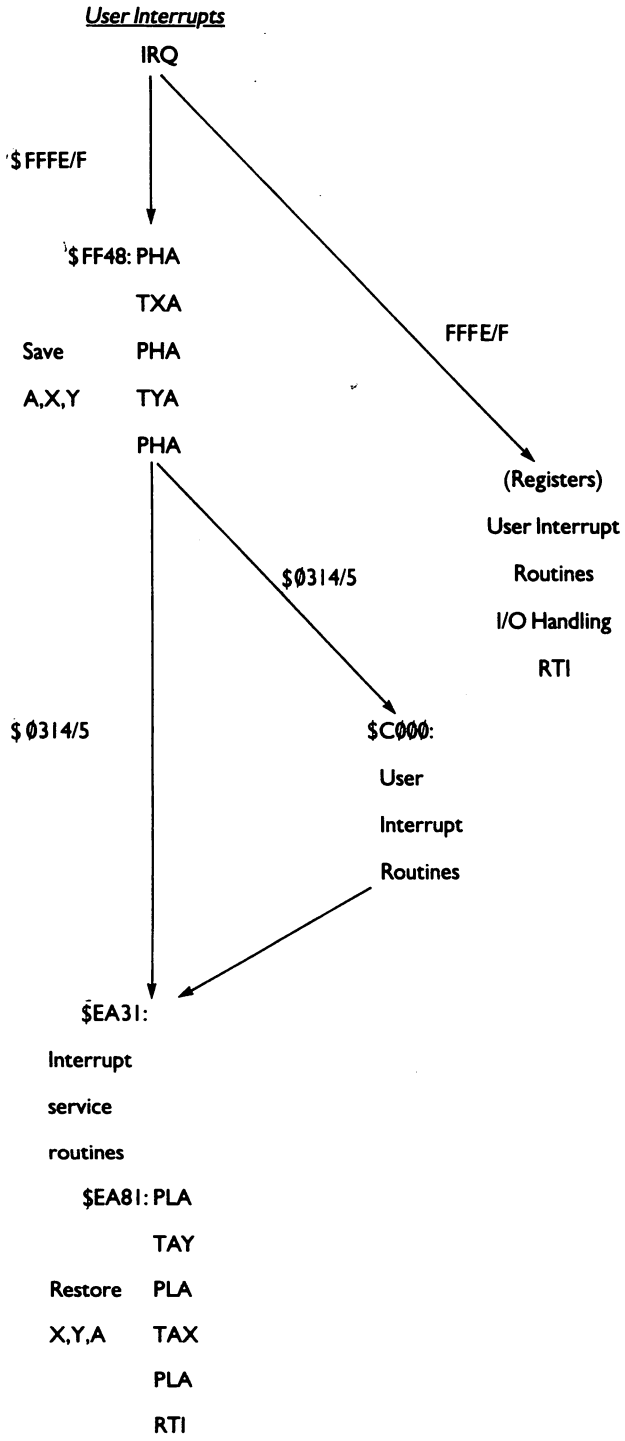
in the monitor, and to switch it off, type:

```
G C200
```

The routine will start running, and control will almost immediately be returned to the user. If the routine is called from BASIC each BRK instruction must be replaced by RTS, to ensure correct return of control, and the routine switched on and off by the appropriate SYS call. Such interrupt-driven routines have distinct advantages. They run as background jobs, freeing the machine for other tasks, and also allow easy implementation of multi-processing, with the interrupt routines and foreground jobs communicating with each other.

Using interrupt routines greatly simplifies the implementation of real-time routines, and of deeply interleaved programs involving a number of different synchronised tasks, as everything is driven by the system clock. Interrupt service routines allow easy debugging and testing of programs, as the values of the associated locations can be changed from BASIC, or the monitor, whilst the program is running.

Using raster interrupts allows screen animation to be synchronised with


User Interrupts
IRQ
$FFFE/F
$FF48: PHA
TXA
Save PHA
A,X,Y TYA
PHA
FFFE/F
(Registers)
User Interrupt
Routines
I/O Handling
RTI
$0314/5
$0314/5
$C000:
User
Interrupt
Routines
$EA31:
Interrupt
service
routines
$EA81: PLA
TAY
Restore PLA
X,Y,A TAX
PLA
RTI

the television raster scan, so that all movement occurs when the raster is off screen (i.e. screen RAM is not updated whilst the video chip is reading it). This completely eliminates the annoying flicker that limits the speed of BASIC animation. Other raster interrupt routines allow us to check how far down the screen the television raster scan has proceeded, allowing split screen work (e.g. both text and graphics) and also enabling us to increase the number of sprites simultaneously available on the screen to seventy-two. Special effects such as automatic motion or rotation of sprites can also be achieved.

The use of interrupts enables us to effectively change the architecture of our machine. We can have a computer that plays music whilst we are writing a business program, or even one with an unlimited number of VIC chips, as we will see in Chapter 8!

Assembler to Hexadecimal

To translate Assembly Code into a form that can be POKEd into memory from BASIC without an assembler is tedious and probably not worth while but, as an example, the following routine is shown assembled in this fashion on the right.

SEI		78		
LDA	#$00	A9	00	
STA	$0314	8D	14	03
LDA	#$C0	A9	C0	
STA	$0315	8D	15	03
CLI		58		
BRK		00		

Note that the low byte of each address is stored first, and that some instructions occupy one byte of memory, whilst others need two or three. This can depend on the addressing mode. To perform this translation for your own program you need to refer to the table showing the hexadecimal code for each mnemonic in each possible addressing mode given in Appendix D, (e.g. LDA in immediate mode=$A9). Be particularly careful with branches and remember that they use one-byte relative addresses, so that, for example:

C00A: BNE $C012

is assembled into the code:

D0 06

which results in the meaning 'jump forward six bytes from the end of the instruction'.

Hexadecimal to BASIC

The following BASIC routine creates a BASIC program from a set of hexadecimal data or a machine code program in memory. If you have not got a disc drive then change each PRINT# command to PRINT and miss out the rest of the disc commands. Run the program so that it outputs to the screen, and then move the cursor onto the top line (60000), press RETURN and the line is included in your program. Continue until you have included all the lines on the screen. (This will only work provided you don't overfill the screen).

```
1 REM"■=SAVE"@0:SLOAD",8
5 INPUT"FILENAME";A$
10 INPUT"START ADDRESS";A
20 INPUT"END ADDRESS";B
30 OPEN 2,8,2,"0:EXEC"+A$+",SEQ,W"
35 PRINT#2,"60000 FORI="+STR$(A)+"TO"+ST
R$(B)+":READA:POKEI,A:NEXT"
40 C=60010
50 A$=STR$(C)+" DATA"
55 PRINTC,A
60 A$=A$+STR$(PEEK(A))+",":A=A+1
70 IFA>BTHEN100
80 IFLEN(A$)<77THEN60
90 A$=A$+STR$(PEEK(A)):PRINT#2,A$:A=A+1:
C=C+10:GOTO50
100 A$=A$+STR$(PEEK(A)):PRINT#2,A$:CLOSE
2
```

Finally, a word on compilers. A compiler is a program that will translate a BASIC program into machine code. Abacus Software's Tiny BASIC compiler also provides an excellent way of learning machine language and will do all the hard work of compiling floating point arithmetic. See also the review of Petspeed in Appendix B.

2 Sound and Synthesisers

In this chapter we discuss the effects that the twenty-nine registers in the SID chip can be used to create and implement a synthesiser program, driven from the Commodore 64 keyboard, utilising all the facilities of SID, including a couple not mentioned in the *Reference Manual.*

The SID Chip

The SID chip registers may be split into five groups, as shown in the table overleaf. The first three groups of seven bytes control the frequency, pulse width, mode and envelope shape for each of the three available voices. The fourth and fifth blocks consist of four bytes each. Of the fourth the first two registers control filter frequency, the third filter mode and resonance, and the fourth volume and filter shape. All the above registers are *write only*, that is they must be POKEd to control the synthesiser, but PEEKing these addresses generates meaningless results. Thus any current register values needed at a later date must either be saved as variables or stored elsewhere in memory. For instance, in the 'Synth' program given later, CR is used for the control register, RF for resonance and filter, and MV for volume and mode. The last four bytes are *read only*. The first two read the values of paddle potentiometers connected to the control ports, and the last two give the current values of oscillator 3 (OSC3) and its envelope shape (ENV3). These can be used to dynamically control the three voices.

The SID Chip: Registers

Register	Address		Function
Voice 1	Hex	Decimal	
0	D400	54272	Frequency low byte
1	D401	54273	Frequency high byte
2	D402	54274	Pulse width low byte
3	D403	54275	Pulse width high nybble
4	D404	54276	Control register (CR)
5	D405	54277	Attack and decay nybbles
6	D406	54278	Sustain and release nybbles
Voice 2			
7	D407	54279	Frequency low byte
8	D408	54280	Frequency high byte
9	D409	54281	Pulse width low byte
10	D40A	54282	Pulse width high nybble
11	D40B	54283	Control register (CR)
12	D40C	54284	Attack and decay nybbles
13	D40D	54285	Sustain and release nybbles
Voice 3			
14	D40E	54286	Frequency low byte
15	D40F	54287	Frequency high byte
16	D410	54288	Pulse width low byte
17	D411	54289	Pulse width high nybble
18	D412	54290	Control register (CR)
19	D413	54291	Attack and decay nybbles
20	D414	54292	Sustain and release nybbles
Filter			
21	D415	54293	Filter frequency low byte
22	D416	54294	Filter frequency high nybble
23	D417	54295	Resonance and filter (RF)
24	D418	54296	Mode and volume (MV)
Read Only			
25	D419	54297	Paddle 1
26	D41A	54298	Paddle 2
27	D41B	54299	Oscillator (OSC3)
28	D41C	54300	Envelope (ENV3)

The SID Chip: Detail of Voice 1 Registers

Register

- 0 Frequency low byte
- 1 Frequency high byte } 16 Bits
- 2 Pulse width low byte
- 3 Pulse width high nybble } 12 Bits
- 4 Control Register (CR)
  - Bit 7 Noise on
  - 6 Pulse on
  - 5 Sawtooth on
  - 4 Triangle on
  - 3 Test bit (if noise locks)
  - 2 Ring modulator on } Synchronised to
  - 1 Synchronisation on } Voice 3
  - 0 Envelope trigger
- 5 Bits 4 – 7 Attack
  - Bits 0 – 3 Decay
- 6 Bits 4 – 7 Sustain
  - Bits 0 – 3 Release

Details of Filter Controls

- 21 Filter frequency low byte
- 22 Filter frequency high nybble } 11 Bits
- 23 Bits 4 – 7 Resonance
  - 3 External
  - 0 – 2 Filter Voices 1 – 3
- 24 Bits 7 Voice 3 off
  - 6 High pass filter on
  - 5 Band pass filter on
  - 4 Low pass filter on
  - 0 – 3 Volume

N.B. The groups of registers for the three voices are identical except that where Voice 1 is synchronised with Voice 3, Voice 2 is synched to Voice 1 and Voice 3 to Voice 2.

Notes and Frequency

To play a note on, say, Voice 1, the frequency registers must first be set. If the required frequency is FOUT then the following piece of BASIC will scale FOUT to the units required by SID, split it into its low and high bytes, and then POKE them into the appropriate registers.

```
FN = FOUT/0.06097
FHI = FN/256
FLO = FN AND 255
SID = 54272
POKE SID, FLO
POKE SID+1, FHI
```

If the required note is to be specified by its musical name then the following code will give the required value of FOUT:

$$FOUT = 2 \uparrow (4 + OCT + N/12)$$

where OCT is the octave number (0 – 7), and N is the note number (0 – 11), according to the following table:

0	C	6	F#
1	C#	7	G
2	D	8	G#
3	D#	9	A
4	E	10	A#
5	F	11	B

This assumes that Middle C is at 256 Hz (cycles per second). The formula is based on the relationship between consecutive octaves (2:1) and consecutive semitones ($2 \uparrow (1/12):1$). Thus the frequency of the C above Middle C is 512 Hz, and there are twelve semitones in an octave. The key of C (the white notes on a piano) consists of the sequence of notes:

. . . C D E F G A B C . . .

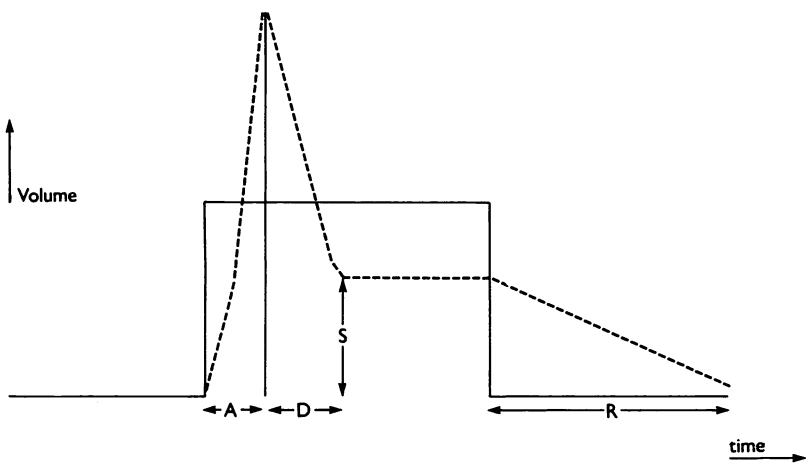
These are the notes that are harmonious to the ear when played together with the base note, C. Two notes sound in harmony if the ratios of their frequencies can be written as the ratio of two simple integers, but you will note that the ratios of the notes in our sequence all contain twelfth roots of two! Nevertheless, in each case a very close approximation to a harmonious ratio can be found. For example:

$$\begin{aligned} G/C &= 2 \uparrow (7/12) \\ &= 1.4983 \ldots \\ &\simeq 3/2 \end{aligned}$$

This is called a fifth interval. In the 'Synth' program presented later, a table of low and high byte values, generated as shown above, is used for simplicity in storing note values.

Envelope Shape

The next step in playing a note is to set the envelope shape of the waveform. This has as much effect on the sound of a note as its frequency and, together with the timbre or overtone structure of the note, governs the instrument with which we associate a particular sound. SID gives the user control over attack and decay rates, sustain levels and the release rate, as in the following diagram.



The solid line represents the time for which, say, a piano key was depressed, and the dashed line the output volume. Initially, the piano string vibrates rapidly with a sudden burst of energy which, almost as quickly, dies down to a steady level, until the piano key is released and the note slowly fades out. The rise time of the note is given by its Attack rate, A, the fall time by the Decay rate, D, the volume by the Sustain level, S, and the time it takes to die out by the Release time, R. The appropriate values must be POKEd into the high and low bytes of registers 5 and 6 of the SID chip. The following expressions show the process:

```
POKE SID + 5, A*16 + D
POKE SID + 6, S*16 + R
```

Each of the parameters A, D, S and R is in the range 0 – 15. By setting the attack rate high (slow) and the rest of the parameters low we could achieve the effect of a piano note played backwards, such is the flexibility of the envelope commands.

Timbre

The timbre, overtone structure, or frequency spectrum of a note is its other major property. The way this is synthesised on the 64 is simple but effective. Three waveforms are available (other than noise). The triangular waveform contains only the odd harmonics (multiples) of the fundamental frequency of a note, and these are inserted in inverse proportion to the square of the harmonic number (multiple). The sawtooth waveform contains the odd harmonics in linear proportion to the reciprocal of the harmonic number. The pulse or square wave contains all the harmonics of the fundamental frequency, their proportion depending on the mark-space ratio (pulse width). By mixing these waveforms, and then filtering the combination, almost any overtone structure you desire can be achieved.

For Voice 1, the fundamental structure of the note is defined by setting bits 4 – 7 in register 4. Bits 1 and 2 control synchronisation and ring modulation. Voices 1, 2 or 3 are filtered by setting bits 0, 1 or 2, respectively, of register 23. Bits 4 – 7 hold a number from 0 to 15, which governs the resonance of the filter. Bits 4, 5 and 6 of register 24 switch on lowpass, bandpass and highpass filtering respectively, in any combination. The resonance governs the bandwidth of the bandpass filter (amount of medium frequency sound transmitted, 0 giving the least and 15, most) and the rolloff of the other two filters. Bits 0 – 3 hold a number between 0 and 15 governing the overall volume of the synthesiser. The filter frequency is an eleven bit number held in registers 21 and 22. Finally, if the pulse waveform is set, the pulse width can be set to any number between 0 and 4096 with the low nybble in register 2 and the high byte in register 3.

Playing a note

To actually play a note it is essential to do three things. Firstly, if you have not aleady done so, switch the gate of the control register off. For Voice 1 this is done with:

```
POKE SID+4, PEEK (SID+4) AND 254
```

Secondly, switch the gate on. This will start the note. It is equivalent to striking the piano key:

```
POKE SID+4, (PEEK (SID+4) AND 254) OR 1
```

This will start the attack/decay/sustain cycle of the note. Finally switch the gate off again to start the release cycle (release the piano key).

Ring modulation

This effect can be used to create percussion sounds, bell tones, chimes and similar effects. It is a technique well known to electric guitarists and made

famous by the late Jimi Hendrix. Ring modulation is a non-harmonic effect. It takes two input signals and outputs the sum and difference frequencies of the input frequencies. Notes that sound good together are related by ratios of simple integers (for example, the relationship between the frequencies of the notes C and G, a fifth, is 3/2). Two consecutive octaves have the frequency ratio 2/1. Music essentially works on a logarithmic scale (to the base 2), the note C in consecutive octaves having frequencies 128, 256, 512 Hz (cycles per second) and so on. Ring modulation, however, is a linear effect. If the two input frequencies are 200 Hz and 300 Hz the output frequencies are 100 Hz and 500 Hz, the difference and sum frequencies. If one input signal is held at a fixed frequency and the other one is stepped up the normal scale, the output signal (provided it is in the audible range of 20 Hz to 20 kHz) will consist of two signals, one stepping up a non-harmonic scale, and the other stepping down! The scale is non-harmonic because the difference in frequency of two semitones in the musical scale is related to the position of the two notes in the scale. The ratio of two consecutive semitones is always given by $2 \uparrow (1/12)$. For instance, the seventh semitone after C is G and $2 \uparrow (7/12) = 1.4983 = 3/2$.

It is this superposition of a linear effect upon a logarithmic scale that gives rise to the non-harmonic effects mentioned above. Note that ring modulation only works with the triangle waveform set on the SID chip.

Dynamic effects

During the period in which the note is actually playing, any of the parameters can be changed. Thus, rapidly varying the volume of the note gives a tremolo effect, rapidly varying the frequency up and down gives vibrato. (Note that the tremolo arm of a guitar actually produces vibrato.) A slow change in frequency gives glissando. Changing the central frequency of the bandpass filter up and down the frequency spectrum produces the 'wahwah' effect well known to electric guitarists. Varying the width of the pulse waveform gives a form of the famous 'phasing' effect, which was originally produced by playing two identical tape recordings slightly out of phase. Varying the pulse width rapidly gives a remarkable non-harmonic effect. Lastly, synchronising Voice 1 to the fundamental frequency of Voice 3 or setting ring modulation (and triangle waveform) and sweeping the frequency of Voice 3 slowly up and down produces a respectable imitation of the currently fashionable 'flanger' (which is actually a swept comb filter). Remember to switch Voice 3 off (unset bit 7 of register 24) when using synchronisation or ring modulation. Synchronisation bases the fundamental waveform of Voice 1 on Voice 3, effectively ANDing the two waveforms together. Many of these effects can be used simultaneously, although the 'Synth' program described below limits you

to a single dynamic effect at a time since BASIC is too slow to write a general program to allow mixing of all effects.

The simplest way to achieve dynamic effects is to increment a counter and POKE its value into, say, the filter frequency register, with the bandpass filter set, whilst the note is playing. This would produce the wahwah effect mentioned above. A more efficient method, particularly in machine code, is to use the oscillator and envelope values of Voice 3, (OSC3 and ENV3) available from the last two registers in the SID chip. In the 'Synth' program below we use OSC3 to obtain tremolo and vibrato for Voice 1 and ENV3 for glissando, phasing, wah-wah and flange. All that is necessary is to PEEK the appropriate register (27 or 28) and POKE the result (or some value based on that result) into the appropriate register (volume, frequency, etc.) while the note is playing. Thus to obtain wah-wah we use:

```
POKE SID + 22, PEEK SID + 28
```

which uses the envelope shape of Voice 3 to sweep the filter frequency up and down the frequency spectrum.

Noise

Noise can *not* be mixed with any of the other waveforms, for any particular voice, using the SID chip. If this is tried the noise waveform will 'lock out', until the test bit for that particular voice is set. White noise consists of all frequencies in the spectrum at once. Filtering produces 'pink' noise. Dynamically sweeping the filter frequency produces a dramatic rushing effect. Rapid attack and decay gives a cymbal-type sound which can be used in a rhythm backing.

A Keyboard-driven synthesiser

The above description is, in parts, brief, as the use of all the effects are illustrated in the synthesiser program below, and also because we do not wish to reproduce material which is presented in the *Reference Manual*. 'Synth' is a keyboard-driven monophonic (one voice) synthesiser emulator. It is intended to be used to experiment with sound effects, which can then be used with the interrupt-driven music interpreter which is unveiled in the next chapter.

It has certain limitations, which do not hinder its use in this manner, and are caused by the fact that it is implemented in BASIC. Firstly, the keyboard is not particularly responsive. This is caused by the fact that computer keyboards are intended to transmit information to the computer about which key has been hit, and not about how long that key has been held down. However, we wish to emulate a musical keyboard and therefore wish to know not only when a key is hit, but also when it is released, so that we

know when to start the release cycle of a note. To get a character from the keyboard (without having to press the RETURN key) we normally have to use a loop like this:

```
100 GET A$: IF A$ = ""THEN 100
```

This says 'look at the keyboard and see if a key has been pressed. Put the contents of the keyboard buffer into A$. If this value is null (no character) try again'. GET actually waits 50 milliseconds between attempts. However, we also wish to test when a key is released. The following piece of code:

```
10 POKE 650,255: REM SET AUTOREPEAT
100 GETA$: IF A$="" THEN 100
110 POKE SID+4,16+1
120 GET B$: IF B$=A$ THEN 120
130 POKE SID+4,16
```

would be expected to gate the triangle waveform of Voice 1, and then start the release cycle when the key is released (i.e. when B$ is no longer equal to A$). Unfortunately, though setting autorepeat on all keys puts a continuous string of characters into the keyboard buffer (try pressing one key after performing this POKE), there is less time between consecutive calls to GET B$ than there is between characters. This means that B$ often gets null characters and skips out of the loop at line 120 (try PRINTing out B$).

We resort to using PEEK (203) (note the error in the *Reference Guide*) which gives the current key pressed (64 for null character). The above piece of code can be replaced by:

```
10 POKE 650,255: REM SET AUTOREPEAT
100 AD=PEEK(203):GET A$:IF AD=64 THEN 100
110 POKE SID+4,16+1
120 BD=PEEK(203):GET A$:IF AD=BD THEN 120
130 POKE SID+4,16
```

The GET A$ statement acts to stop the text screen being corrupted by unwanted characters. The values of PEEK (203) are a bit obscure (see program), due to the way the keyboard is scanned.

Secondly, when dynamic effects are used, it becames necessary to perform POKE commands within the loop at line 120. This means that it becomes very difficult to write a general program to mix dynamic effects in BASIC. Thus in 'Synth', we may only have one dynamic effect at once, and there is a separate piece of code for each one.

Further, because we regularly have to test for a character, some of the dynamic effects are a bit noisy with some SID settings. This would not occur when using the effects from a program or in machine code, but is entirely caused by the fact that 'Synth' is a keyboard-driven BASIC program. On my computer, tremolo is also rather noisy on the triangle setting because the SID chip makes a slight click whenever the volume is changed. But then

I have a very early machine (it has grey function keys). By the time this book is published Commodore should have released their own high-quality 3½ octave music keyboard for the 64 which should cure the problems mentioned above.

The other limitations of our program are caused by the SID chip itself. The noise waveform cannot be mixed with the others for any particular voice – however, when playing polyphonic music with more than one voice (see next chapter) noise can be set on one or more voice with other waveforms on the other voices. Ring modulation can also only be used with the triangle waveform.

Finally, no control is provided in the program for changing the ADSR of Voice 3, which controls most of the dynamic effects. We leave this as an exercise. For instance, try changing the attack rate of the phasing effect in line 641 of 'Synth' from 12 to 15. This changes the sound from a remarkable non-harmonic effect to the smooth phase normally associated with this sound.

The 'Synth' program

Type in the program below. Note that it is presented in four sections, each with a complete line by line description. It is easy to take out any subroutines you need and insert them into your own programs, but remember that you will incur all the problems of interleaving and synchronisation associated with them. It would be better to use the interrupt-driven music interpreter given in the next chapter. Remember also to make sure that the variables MV, CR, RF, A, D, S, R always contain the required values of the associated registers and are POKEd into them at the appropriate times.

Initially it is not necessary to implement the dynamic effects, so lines 400 to 668 and 3000 to 3160 can be omitted and typed in when the rest of the program is debugged. If you do this, a dummy line 3100 RETURN must be inserted to stop the programe crashing on initialisation. Full instructions for using the program are given later. Make sure that you follow the line by line description of what the program does as you are keying it in. This will teach you more about how to use the SID chip, and the 64 in general, than any manual can.

'Synth': Section 1 Description

Lines 10–500 initialise and scan keyboard:

Line	10	Clear SID chip
	20	Title
	30	Set up default ADSR

40 Set up: RF resonance and filter
MV mode and volume (volume = 15 and Voice 3 off)
CR control register (set triangle)
ADSR attack, decay, sustain, release
DY = 0 no dynamic effects
1 tremolo
2 vibrato
4 glissando
8 phasing
16 wah-wah
32 flanging
DE depth of dynamic effects
50 Set autorepeat on all keys. Set up MV register
60 Set up W (pulse width)
70 Initialisation routines
100 Get a character. Wait 50 ms and try again.
110/124 Keys defined (notes C3 to C5)
125 DEL character is note D5
130 END
200/370 Controls
380 Dynamic effect?
390 Not a valid character: try again
400/500 Which dynamic effect?

'Synth': Section 1 of Program

```
1 REM"■=SAVE"@0:SYNTH",8
10 SID=54272:FORL=SIDTOSID+24:POKEL,0:NE
XT
20 PRINT"■■■■■■■■■■■■■■■■■■■■■■*** SYNTH ***"
30 POKESID+5,5*16+8:POKESID+6,12*16+3
40 RF=15*16:MV=15+128:CR=16:A=5:D=8:S=12
:R=3:DY=0:DE=0.5
50 POKE650,255:POKESID+24,MV
60 W=50:GOSUB2071
70 GOSUB4010
100 AD=PEEK(203):GETA$:IFAD=64THEN100
110 IFAD=57THENHF= 8:LF= 97:GOSUB1000:GO
TO800:REM ←
111 IFAD=56THENHF= 9:LF=104:GOSUB1000:GO
TO800:REM 1
112 IFAD=59THENHF=10:LF=143:GOSUB1000:GO
TO800:REM 2
```

```
113 IFAD= 8THENHF=11:LF= 48:GOSUB1000:GO
TO800:REM 3
114 IFAD=11THENHF=12:LF=143:GOSUB1000:GO
TO800:REM 4
115 IFAD=16THENHF=14:LF= 24:GOSUB1000:GO
TO800:REM 5
116 IFAD=19THENHF=15:LF=210:GOSUB1000:GO
TO800:REM 6
117 IFAD=24THENHF=16:LF=195:GOSUB1000:GO
TO800:REM 7
118 IFAD=27THENHF=18:LF=209:GOSUB1000:GO
TO800:REM 8
119 IFAD=32THENHF=21:LF= 31:GOSUB1000:GO
TO800:REM 9
120 IFAD=35THENHF=22:LF= 96:GOSUB1000:GO
TO800:REM 0
121 IFAD=40THENHF=25:LF= 30:GOSUB1000:GO
TO800:REM +
122 IFAD=43THENHF=28:LF= 49:GOSUB1000:GO
TO800:REM -
123 IFAD=48THENHF=31:LF=165:GOSUB1000:GO
TO800:REM £
124 IFAD=51THENHF=33:LF=135:GOSUB1000:GO
TO800:REM"
125 IFAD= 0THENHF=37:LF=162:GOSUB1000:GO
TO800:REM DEL
130 IFAD=14THEN1010:REM E
200 IFAD=10THENGOSUB2000:GOTO100:REM A
210 IFAD=13THENGOSUB2010:GOTO100:REM S
220 IFAD=18THENGOSUB2020:GOTO100:REM D
230 IFAD=17THENGOSUB2030:GOTO100:REM R
240 IFAD=62THENGOSUB2040:GOTO100:REM Q
250 IFAD=29THENGOSUB2050:GOTO100:REM H
260 IFAD=21THENGOSUB2060:GOTO100:REM F
270 IFAD= 9THENGOSUB2070:GOTO100:REM W
280 IFAD=42THENGOSUB2080:GOTO100:REM L
290 IFAD=22THENGOSUB2090:GOTO100:REM T
300 IFAD=12THENGOSUB2100:GOTO100:REM Z
310 IFAD=41THENGOSUB2110:GOTO100:REM P
320 IFAD=31THENGOSUB2120:GOTO100:REM V
330 IFAD=28THENGOSUB2130:GOTO100:REM B
340 IFAD=39THENGOSUB2140:GOTO100:REM N
350 IFAD=36THENGOSUB2150:GOTO100:REM M
360 IFAD=23THENGOSUB2160:GOTO100:REM X
```

```
370 IFAD=26THENGOSUB2170:GOTO100:REM G
380 IFAD=54THEN400:REM ↑
390 GOTO100
400 AD=PEEK(203):GETA$:IFAD=64THEN400:RE
M ""
410 IFAD=22THENGOSUB3000:REM T
420 IFAD=31THENGOSUB3010:REM V
430 IFAD=26THENGOSUB3020:REM G
440 IFAD=41THENGOSUB3030:REM P
450 IFAD= 9THENGOSUB3040:REM W
460 IFAD=21THENGOSUB3050:REM F
500 GOTO100
```

Synth: Section 2 Description

Lines 600–1000 play the note whilst the key is pressed:

600–605 Check which dynamic effect

610–618 Tremolo: add OSC3 to volume

610 Extract volume from MV (bottom four bits) and scale by 2/3
Gate control register to start note (POKE with CR+1)
Set Voice 3 to triangle wave (bit 4)

612 Add value of Voice 3 (OSC3) to volume after scaling into range (0 – 5). (Volume always <=15.)

614 Put new volume into MV register (lower nybble from VQ and upper nybble from MV)

616 Get character from buffer. If still the same as last character then return to 612. Otherwise:

618 POKE control register with CR to finish note

620–628 Vibrato: add OSC3 to Voice 1 frequency

620 Form frequency from HF and LF. Set Voice 3 to triangle wave and gate control register (start note)

622/624 Add scaled value of Voice 3 (OSC3) to frequency FQ
Put high byte of FQ into low byte of HF and low byte into LF. Remember that an integer has sixteen bits, though we normally only consider the lower byte of the two

626 Is key still depressed? If yes then 622. Otherwise:

628 Finish note

630–638 Glissando: add ENV3 to Voice 1 frequency

630 Form frequency and set Voice 3 to triangle. Note that Bit 1 of Voice 3 control register must be set to 0 before it is gated

631 Set ADSR of Voice 3 (changing attack alters speed of glissando). Gate Voices 1 and 3

632/634 Add scaled value of envelope 3 to Voice 1. Split into high and low bytes and POKE into SID chip

636 Check key is still depressed

638 Switch Voice 1 off and GOTO 100

640–648 Phasing: pulse width controlled by ENV3

640/641 Set ADSR of Voice 3 and gate Voices 1 and 3 (changing attack rate alters speed of phasing)

642/644 Split scaled value of ENV3 into high and low bytes and set pulse width

646 Check if key depressed

648 Ungate Voice 1

650–658 Wah-wah: filter frequency controlled by ENV3

650/652 Set ADSR of Voice 3 and gate Voices 1 and 3 (Changing attack alters speed of wah-wah)

654 POKE scaled value of ENV3 into high byte of filter frequency.

656/658 If key not depressed then ungate Voice 1

660–668 Flanging: Voice 3 frequency controlled by ENV3

660/663 Set ADSR of Voice 3 and gate Voices 1 and 3 (Changing attack alters flange rate)

664 Poke low byte of Voice 3 frequency with ENV3

666/668 If key no longer depressed then finish

800–840 No dynamic effects.

800 If dynamic effect required GOTO 600 to find out which one

805 Start note

810 Wait long enough for buffer to fill. If key still depressed then wait some more

820 If note then switch Voice 1 off

840 GOTO 100

1000 Subroutine to set Voice 1 frequency

1010 If E was depressed then clear SID chip and return to BASIC.

'Synth': Section 2 of Program

```
600 IFDY=1THEN610
601 IFDY=2THEN620
602 IFDY=4THEN630
603 IFDY=8THEN640
604 IFDY=16THEN650
605 IFDY=32THEN660
610 VR=(MVAND15)*2/3:POKESID+4,CR+1:POKE
SID+18,16
612 VQ=VR+PEEK(SID+27)*DE/25
614 POKESID+24,(VQAND15)OR(MVAND240)
616 BD=PEEK(203):GETA$:IFAD=BDTHEN612
618 POKESID+4,CR:GOTO100
620 FR=HF*256+LF:POKESID+18,16:POKESID+4
,CR+1
622 FQ=FR+PEEK(SID+27)*DE:HF=INT(FQ/256)
624 LF=FQAND255:POKESID+0,LF:POKESID+1,H
F
626 BD=PEEK(203):GETA$:IFAD=BDTHEN622
628 POKESID+4,CR:GOTO100
630 FR=HF*256+LF:POKESID+18,16
631 POKESID+19,12*16+15:POKESID+20,240:P
OKESID+18,17:POKESID+4,CR+1
632 FQ=FR+PEEK(SID+28)*DE:HF=INT(FQ/256)
634 LF=FQAND255:POKESID+0,LF:POKESID+1,H
F
636 BD=PEEK(203):GETA$:IFAD=BDTHEN632
638 POKESID+4,CR:POKESID+18,16:GOTO100
640 POKESID+4,CR+1:POKESID+18,16
641 POKESID+19,12*16+15:POKESID+20,240:P
OKESID+18,17
642 WQ=10+PEEK(SID+28)*DE*32:J=INT(WQ/25
6)
644 K=WQAND255:POKESID+3,J:POKESID+2,K
646 BD=PEEK(203):GETA$:IFAD=BDTHEN642
648 POKESID+4,CR:POKESID+18,16:GOTO100
650 POKESID+18,16
652 POKESID+19,12*16+15:POKESID+20,240:P
OKESID+18,17:POKESID+4,CR+1
```

```
654 POKESID+22,128+PEEK(SID+28)/2
656 BD=PEEK(203):GETA$:IFAD=BDTHEN654
658 POKESID+4,CR:POKESID+18,16:GOTO100
660 POKESID+18,16
662 POKESID+19,12*16+15:POKESID+20,240:P
OKESID+18,17:POKESID+4,CR+1
664 POKESID+14,PEEK(SID+28):REM LOW BYTE
666 BD=PEEK(203):GETA$:IFAD=BDTHEN664
668 POKESID+4,CR:POKESID+18,16:GOTO100
800  IFDY<>0THEN600
805 POKESID+4,CR+1
810 BD=PEEK(203):GETA$:IFAD=BDTHEN810
820 POKESID+4,CR
840 GOTO100
1000 POKESID,LF:POKESID+1,HF:RETURN
1010 FORL=SIDTOSID+24:POKEL,0:NEXT:END
```

'Synth': Section 3 Description

Lines 2000–2173 set control registers and parameters.

2000–2039 Set Voice 1 ADSR

2040–2041 Set resonance if Q was depressed. Subroutine 4000 tabs to middle of page. Inverse Q is 'cursor down'. Put Q into high byte of RF and POKE into register

2050–2054 Toggle highpass filter on and off (H)

2050 Invert bit 6 of MV and POKE into register. (Clue: 64 + 191 = 255)

2051/2054 Extract bits 4, 5 and 6 from MV (112 = 2 ↑ 4 + 2 ↑ 4 + 2 ↑ 6). If no filtering then set lowest bit of RF to 0, else set to 1, and set register to switch Voice 1 filtering on or off as appropriate. Print OFF or ON.

2060–2061 Get filter frequency and scale from Hertz (only high byte used)

2070–2073 Get pulse width, scale from percentage, split into low and high bytes

2080–2084 Toggle lowpass filter on and off, invert bit 4 of MV register. (See lines 2051–2054 for further details)

2090–2093 Toggle triangle waveform on and off.

2090 Check if noise set (top bit of CR) and switch off if necessary. (If noise is switched on at the same time as T, Z or P, SID will lock up until the test bit of CR is set!)

2091/2093 Toggle bit 4 of CR register

2100–2103 Toggle sawtooth waveform (bit 5 of CR)

2110–2113 Toggle pulse waveform (bit 6 of CR)

2120 Set volume if V was depressed.

2121 Put volume into lower nybble of MV

2130–2134 Toggle bandpass filter (invert bit 5 of MV register). See description, lines 2051–2054

2140–2145 Set noise waveform for Voice 1

2140/2142 If Z, T or P waveform set – switch them off!

2143 Toggle bit 7 of CR

2150–2152 Toggle ring modulator on or off.

2150 Invert bit 2 of CR

2160–2162 Toggle synch (invert bit 1 of CR)

2170–2173 Set Voice 3 frequency if G pressed

2171 Scale G from Hertz, split into low and high bytes

'Synth': Section 3 of Program

```
2000 GOSUB4000:INPUT"[redacted] A ATTACK";A:I
FA<0ORA>15THEN2000
2001 POKESID+5,A*16+D
2009 RETURN
2010 GOSUB4000:INPUT"[redacted] S SUSTAIN";
S:IFS<0ORS>15THEN2010
2011 POKESID+6,S*16+R
2019 RETURN
2020 GOSUB4000:INPUT"[redacted] D DECAY";D:I
FD<0ORD>15THEN2020
2021 POKESID+5,A*16+D
2029 RETURN
2030 GOSUB4000:INPUT"[redacted] R RELEASE"
;R:IFR<0ORR>15THEN2030
2031 POKESID+6,S*16+R
```

```
2039 RETURN
2040 GOSUB4000:INPUT"[redacted] Q RESON
ANCE";Q:IFQ<0ORQ>15THEN2040
2041 RF=(RFAND15)OR(Q*16):POKESID+23,RF:
RETURN
2050 BIT=NOTMVAND64:MV=BIT OR(MVAND191):
POKESID+24,MV
2051 NIB=MVAND112:IFNIB=0THENRF=RFAND254
2052 IFNIB<>0THENRF=(RFAND254)OR1
2053 POKESID+23,RF:IFBIT=0THENPRINT"[redacted]
[redacted] H HIGHPASS NO ":RETURN
2054 IFBIT<>0THENPRINT"[redacted] H HIG
HPASS YES":RETURN
2060 GOSUB4000:INPUT"[redacted] F FREQUE
NCY";F:IFF<30ORF>12000THEN2060
2061 F=(F-30)*256/12000:POKESID+22,F:RET
URN
2070 GOSUB4000:INPUT"[redacted] W PULSE W
IDTH";W:IFW<0ORW>100THEN2070
2071 W=W*40.95:J=INT(W/256):K=W-J*256
2073 POKESID+3,J:POKESID+2,K:RETURN
2080 BIT=NOTMVAND16:MV=BIT OR(MVAND239):
POKESID+24,MV
2081 NIB=MVAND112:IFNIB=0THENRF=RFAND254
2082 IFNIB<>0THENRF=(RFAND254)OR1
2083 POKESID+23,RF:IFBIT=0THENPRINT"[redacted]
[redacted] L LOWPASS  NO ":RETURN
2084 IFBIT<>0THENPRINT"[redacted] L LO
WPASS  YES":RETURN
2090 IF(CRAND128)=128THENGOSUB2140
2091 BIT=NOTCRAND16:CR=BIT OR(CRAND239)
2092 IFBIT=0THENPRINT"[redacted] T TRIANGL
E NO ":RETURN
2093 IFBIT<>0THENPRINT"[redacted] T TRIANG
LE YES":RETURN
2100 IF(CRAND128)=128THENGOSUB2140
2101 BIT=NOTCRAND32:CR=BIT OR(CRAND223)
2102 IFBIT=0THENPRINT"[redacted] Z SAWTOOTH
 NO ":RETURN
2103 IFBIT<>0THENPRINT"[redacted] Z SAWTOOT
H YES":RETURN
2110 IF(CRAND128)=128THENGOSUB2140
2111 BIT=NOTCRAND64:CR=BIT OR(CRAND191)
2112 IFBIT=0THENPRINT"[redacted] P PULSE
```

```
   NO ":RETURN
2113 IFBIT<>0THENPRINT"[redacted] P PULSE
     YES":RETURN
2120 GOSUB4000:INPUT"[redacted] V VOL
UME";V:IFV<0ORV>15THEN2120
2121 MV=(MVAND240)OR(VAND15):POKESID+24,
MV:RETURN
2130 BIT=NOTMVAND32:MV=BIT OR(MVAND223):
POKESID+24,MV
2131 NIB=MVAND112:IFNIB=0THENRF=RFAND254
2132 IFNIB<>0THENRF=(RFAND254)OR1
2133 POKESID+23,RF:IFBIT=0THENPRINT"[redacted]
[redacted] B BANDPASS NO ":RETURN
2134 IFBIT<>0THENPRINT"[redacted] B BAND
PASS YES":RETURN
2140 IF(CRAND16)=16THENGOSUB2090
2141 IF(CRAND32)=32THENGOSUB2100
2142 IF(CRAND64)=64THENGOSUB2110
2143 BIT=(NOTCR)AND128:CR=BIT OR(CRAND12
7)
2144 IFBIT=0THENPRINT"[redacted] N NOISE
NO ":RETURN
2145 IFBIT<>0THENPRINT"[redacted] N NOISE
 YES":RETURN
2150 BIT=NOTCRAND4:CR=BIT OR(CRAND251)
2151 IFBIT=0THENPRINT"[redacted] M RI
NGMOD  NO ":RETURN
2152 IFBIT<>0THENPRINT"[redacted] M R
INGMOD  YES":RETURN
2160 BIT=NOTCRAND2:CR=BIT OR(CRAND253)
2161 IFBIT=0THENPRINT"[redacted] X S
YNCH    NO ":RETURN
2162 IFBIT<>0THENPRINT"[redacted] X
SYNCH    YES":RETURN
2170 GOSUB4000:INPUT"[redacted] G VOIC
E 3 FREQ";G:IFG<0ORG>3906THEN2170
2171 G=G/0.059604645:J=INT(G/256):K=G-J*
256
2173 POKESID+15,J:POKESID+14,K:RETURN
```

'Synth': Section 4 Description

3000–3160	Toggle bits of dynamic effect variable DY. See lines 400–500 of program for different values
4000	Subroutine to move cursor to right hand column
4010–5200	Initialisation subroutine. Voice 1 set to triangle waveform. Rest of screen display printed

'Synth': Section 4 of Program

```
3000 IFDY<>1THENDY=1:GOTO3100
3001 IFDY=1THENDY=0:GOTO3100
3010 IFDY<>2THENDY=2:GOTO3100
3011 IFDY=2THENDY=0:GOTO3100
3020 IFDY<>4THENDY=4:GOTO3100
3021 IFDY=4THENDY=0:GOTO3100
3030 IFDY<>8THENDY=8:GOTO3100
3031 IFDY=8THENDY=0:GOTO3100
3040 IFDY<>16THENDY=16:GOTO3100
3041 IFDY=16THENDY=0:GOTO3100
3050 IFDY<>32THENDY=32:GOTO3100
3051 IFDY=32THENDY=0:GOTO3100
3100 IFDY=1THENPRINT"↑T
 TREMELO    YES"
3101 IFDY<>1THENPRINT"↑
T TREMELO    NO "
3110 IFDY=2THENPRINT"↑
V VIBRATO    YES"
3111 IFDY<>2THENPRINT"
↑V VIBRATO    NO "
3120 IFDY=4THENPRINT"
↑G GLISSANDO YES"
3121 IFDY<>4THENPRINT"
↑G GLISSANDO NO "
3130 IFDY=8THENPRINT"
↑P PHASING    YES"
3131 IFDY<>8THENPRINT"
↑P PHASING    NO "
3140 IFDY=16THENPRINT"
↑W WAA WAA    YES"
3141 IFDY<>16THENPRINT"
↑W WAA WAA    NO "
3150 IFDY=32THENPRINT"
↑F FLANGING   YES"
```

```
3151 IFDY<>32THENPRINT"[redacted]
[redacted]F FLANGING  NO "
3160 RETURN
4000 PRINT"[redacted]";:RETURN
4001 PRINT"[redacted]";:RETURN
4010 GOSUB2051:GOSUB2081:BIT=1:GOSUB2092
:BIT=0:GOSUB2102:GOSUB2112:GOSUB2131
4011 GOSUB2144:GOSUB2151:GOSUB2161:GOSUB
3100
5000 GOSUB4000:PRINT"[redacted] A ATTACK"
5010 GOSUB4000:PRINT"[redacted] S SUSTAIN"
5020 GOSUB4000:PRINT"[redacted] D DECAY"
5030 GOSUB4000:PRINT"[redacted] R RELEASE"
5040 GOSUB4000:PRINT"[redacted] Q RESON
ANCE"
5060 GOSUB4000:PRINT"[redacted] F FREQUE
NCY"
5070 GOSUB4000:PRINT"[redacted] W PULSE W
IDTH"
5120 GOSUB4000:PRINT"[redacted] V VOL
UME"
5170 GOSUB4000:PRINT"[redacted] G VOIC
E 3 FREQ"
5200 RETURN
```

'Synth' program controls

Keys and Notes

←	C3	8	D4
1	D3	9	E4
2	E3	0	F4
3	F3	+	G4
4	G3	–	A4
5	A3	£	B4
6	B3	HOME	C5
7	C4	DEL	D5

Controls

N	Noise on/off. Noise on switches Z, T and P off
Z	Sawtooth waveform on/off. Z on switches N off.
T	Triangle waveform on/off. T on switches N off

P	Pulse waveform on/off. P on switches N off
B	Bandpass filter on/off
H	Highpass filter on/off } F must be set
L	Lowpass filter on/off
M	Ringmodulator on/off. T and G must be set
X	Synchronisation on/off. G must be set (low!)
^T	Tremolo on/off. G must be set (circa 5 Hz)
^V	Vibrato on/off.
^G	Glissando on/off
^P	Phasing on/off. P must be set
^W	Wah-wah on/off. B, H or L must be set
^F	Flanging on/off. X or M must be set
E	Return to BASIC.

Parameters

Press <RETURN> after entering value of each parameter.

A	Attack (0–15)
D	Decay (0–15)
S	Sustain (0–15)
R	Release (0–15)
W	Pulse width (0–100)
F	Filter frequency (30–1200)
Q	Resonance (0–15)
G	Voice 3 frequency (0–3906)
V	Volume (0–15)

Table of useful settings for 'Synth'

EFFECT	A D S R	ATTRIBUTES ON		OTHER SETTINGS
Piano	0900	Z		
Xylophone	0909	T		
Clarinet	5500	T		
Violin	5859	Z		
Flute	3091	T		
Chimes	0900	TM		G=500
Reed	5555	P		W=3
Synch	0900	PX		W=3, G=64
Cymbals	0999	N		
Tremolo	5555	Z	↑T	G=3
Vibrato	5555	P	↑V	G=7, W=50
Phasing	5555	P	↑P	
Wahwah	5555	ZB	↑W	
Glissando	0900	T	↑G	
Flange1	0900	TX	↑F	G=64
Flange2	0900	TM	↑F	
Boing!	0900	TXM	↑F	G=64
Synth1	0999	TBXM	↑W	G=128
Synth2	0999	TBXM	↑W	G=7
Swoosh	0999	NB	↑W	

Using 'Synth'

When you have entered the program and hit RUN the following display should appear on your screen:

```
                *** SYNTH ***

 N NOISE      NO       A ATTACK
 Z SAWTOOTH   NO       D DECAY
 T TRIANGLE   YES      S SUSTAIN
 P PULSE      NO       R RELEASE
 B BANDPASS   NO       W PULSE WIDTH
 H HIGHPASS   NO       F FREQUENCY
 L LOWPASS    NO       Q RESONANCE
 M RINGMOD    NO       G VOICE 3 FREQ
 X SYNCH      NO       V VOLUME

↑T TREMELO     NO
↑V VIBRATO     NO
↑G GLISSANDO   NO
↑P PHASING     NO
↑W WAA WAA     NO
↑F FLANGING    NO
```

The keys along top of the Commodore 64 keyboard are used to drive the synthesiser. They represent the white keys on a piano from the C below middle C(←) to the D in the octave above middle C (DEL). These values can be easily changed by substituting new values of HF and LF in lines 110 to 125 of 'Synth' from the tables on pages 384/6 of the *Reference Manual*, but note that notes 82, 83 and 84 on page 386 should read D–5, D#–5 and E–5 respectively. The table below gives values for the key of C.

Key of C: Encoding

MUSICAL NOTE				OSCILLATOR FREQUENCY			
Note	Octave	Number	Hex	Decimal	Hi byte	Lo byte	Hex
E	2	36	$24	1351	5	71	$547
F	2	37	$25	1432	5	152	$598
G	2	39	$27	1607	6	71	$647
A	2	41	$29	1804	7	12	$70C
B	2	43	$2B	2025	7	233	$7E9
C	3	48	$30	2145	8	97	$861
D	3	50	$32	2408	9	104	$968
E	3	52	$34	2703	10	143	$A8F
F	3	53	$35	2864	11	48	$B30
G	3	55	$37	3215	12	143	$C8F
A	3	57	$39	3608	14	24	$E18
B	3	59	$3B	4050	15	210	$FD2
C	4	64	$40	4291	16	195	$10C3
D	4	66	$42	4817	18	209	$12D1
E	4	68	$44	5407	21	31	$151F
F	4	69	$45	5728	22	96	$1660
G	4	71	$47	6430	25	30	$191E
A	4	73	$49	7217	28	49	$1C31
B	4	75	$4B	8101	31	165	$1FA5
C	5	80	$50	8583	33	135	$2187
D	5	82	$52	9634	37	162	$25A2
E	5	84	$54	10814	42	62	$2A3E
F	5	85	$55	11457	44	193	$2CC1
G	5	87	$57	12860	50	60	$323C
A	5	88	$58	14435	56	99	$3863
B	5	91	$5B	16203	63	75	$3F4B
C	6	96	$60	17167	67	15	$430F
D	6	98	$62	19269	75	69	$4B45
E	6	100	$64	21629	84	125	$547D

The waveform is set initially to triangle with ADSR as in line 40 of the program. Try changing the waveform by pressing N, Z, T or P and listen to the different sounds. Note that pressing N switches the other three attributes to 'NO'. Try the different instrument settings given in the table. Change the ADSR settings by typing, for instance:

A5 <RETURN>

to set attack rate to 5. All attributes on the top left-hand side of the display require only a single keypress to toggle them on or off, whilst the dynamic effects at the bottom left require '↑' followed by the appropriate character. The parameters on the right-hand side of the screen require a character, then a number in the appropriate range, followed by RETURN.

Now set the parameters for chimes from the tables, and try playing up and down the keys. You should hear two notes, one going up as you play up the keys, as you would expect, and a deep sound which goes down the scale simultaneously! These are the sum and difference frequencies described above. Also try the reed instrument setting with the various values of pulse width. Try the synch settings with different values of Voice 3 frequency (don't set this too high).

The tremolo and vibrato settings from the table only work with values of G around 5 Hz, otherwise BASIC can not keep up. You should be able to hear the rapid changes in volume and frequency respectively. Phasing gives a distinctly non-harmonic effect, but see the comments on ADSR of ENV3 above, and try changing the attack rate of Voice 3 to 15. Flange1 and Flange2 use synchronisation and ring modulation while varying the frequency of Voice 3. Try changing ↑F on and off and listen to the difference. Finally 'Boing!' is a remarkable effect made by mixing the two together. The range of sounds that can be produced is incredible. Do not forget to experiment with the envelope shape rather than the more exciting effects. This has a great effect on the final sound of the note. Extend the table of settings. Whenever you discover an exciting new one write it down. You will then be in a position to make full use of the music interpreter in the next chapter.

3 Music and Sequencers

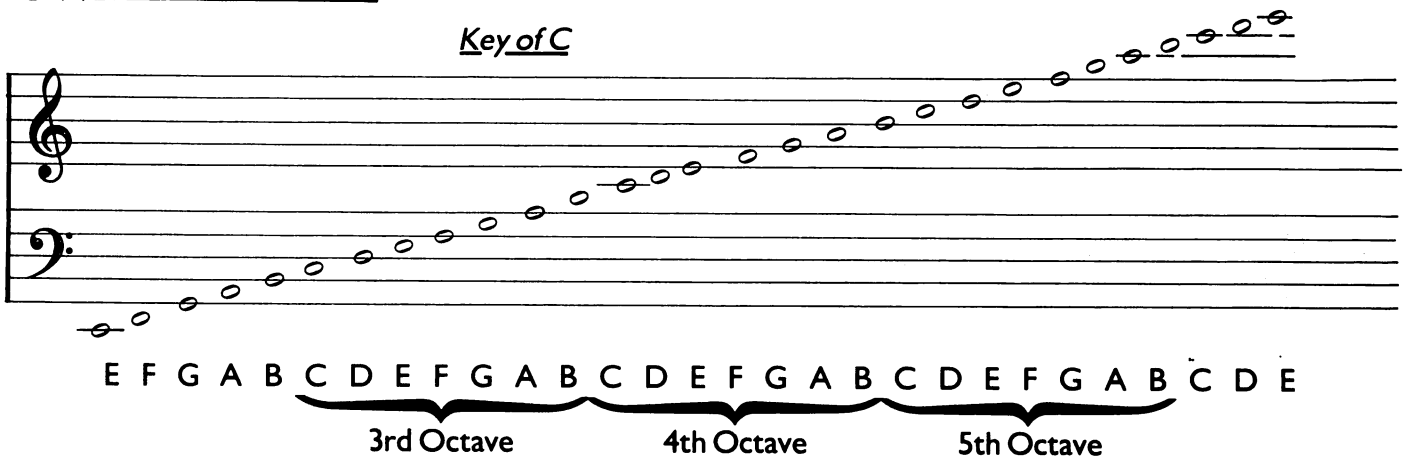
I hope you got 'Synth' working properly. The program in this chapter is a machine code, interrupt-driven, music interpreter. It is far shorter and far more powerful than 'Synth,' although we leave it up to the reader to implement dynamic effects as we reach the stage where the program must be designed around a particular application. You must, of course, have a monitor/assembler to do this. For BASIC programmers we provide a short, albeit rather tedious, BASIC program which POKEs the music interpreter, note tables and music into memory. If you have not yet gathered what this program does, here it is again. 'Roundabout' plays music as a background job on the Commodore 64. Provided you do not do anything else with the interrupt pointers, your Commodore 64 will sit there playing the theme tune to the Magic Roundabout (or any other music you care to code up) while you sit there and get on with writing or running another program, with virtually no degradation in speed of the machine. This is the ideal way to put background music into your programs.

Before we tackle 'Roundabout' however, we start with a brief introduction to musical ideas for programmers. This should take you to the stage where you can code up your own piece of music for use with the program.

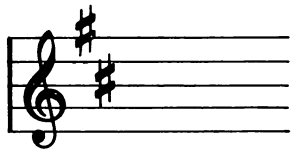
Musical Notation

Musical scores are based on an antique system of notation that bears little relationship to the needs of any particular musical instrument, let alone a Commodore 64. Various attempts have been made to replace this system (e.g. tablature notation for guitarists) but none have been successful. Figure 1 shows four octaves from the key of C on the musical stave with the treble clef symbol at top left, with the bass clef below. The stave is read from left to right, with low notes at the bottom and high notes at the top, and the note positions alternate between the lines and the spaces. Figure 2 gives the symbols for the various note lengths. The musical stave is split horizontally into 'bars', as indicated by the vertical lines. Each bar normally (i.e. in the standard 4/4 time) holds one whole note or the equivalent two minims, four crotchets, etc., or any appropriate combination.

Fig. (1) Musical Note Values



Key Signatures



Key of D:
this means that every C note and every F note, in every octave, is played a semitone higher (i.e. sharp) unless indicated otherwise.

If a sharp (♯) or a flat (♭) appears in a piece of music, then the note preceded by that symbol is played a semitone higher or lower, respectively, for the rest of that bar unless indicated otherwise (usually by the natural sign, ♮).

Fig. (2) Musical Note Lengths.

semibreve (whole note)

minim (half note)

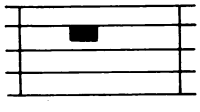
crochet (quarter note)

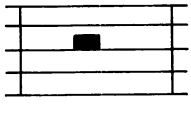
quaver (eighth note)

semiquaver (sixteenth note)

demisemiquaver (thirtysecond note)

Rests (silence)





whole rests
(bar lines indicated)

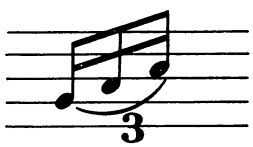
 quarter rest

 eighth rest

 sixteenth rest

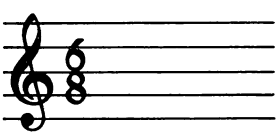
 thirtysecond rest

If any of the above symbols are followed by a dot, then the note length is increased by one half. If three notes are bracketed together thus:



then they are played in the time it normally takes for two equivalent notes (often called a triplet). Sometimes sixtyfourth notes (hemidemisemiquavers) are used!

Time Signatures



means that there are six quavers (eighth notes or equivalent) in each bar.

Keys

In the last chapter we picked out seven notes from the twelve possible semitones to form the key of C. There are twelve possible musical keys that we can play music in, formed by starting from any semitone in an octave and picking out the seven notes above it with the same sequence of intervals as we chose for the key of C. That is:

C tone D tone E semitone F tone G tone A tone B semitone C

A tone is two semitones. For instance the key of G is:

G tone A tone B semitone C tone D tone E tone F semitone G

where every F note in the piece of music is sharpened (raised a semitone). Corresponding to each major key constructed in this way there is a minor key, taking the same key signature as the major key, but with a different

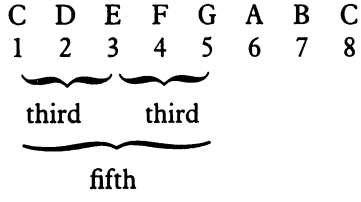
sequence of intervals. For instance, corresponding to C major is the key of A minor:

A tone B semitone C tone D tone E semitone F tone G tone A

Typically, music written in the key of A minor (Am) would start and finish on an A, rather than C, note. Whereas major scales have a bright, extrovert quality, music written in a minor key tends to sound plaintive or introverted.

Chords

In the last chapter we saw that two notes, such as C and G (forming an interval called a fifth), sounded good together because their frequencies were in the simple ratio 3:2. We now split the fifth up into two smaller intervals, called thirds, like this:

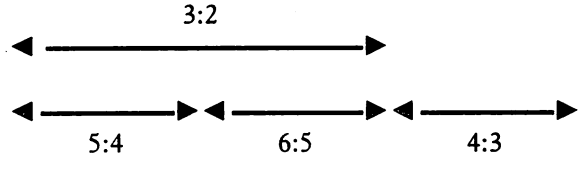


We can include the three notes C, E and G in a set which constitute the chord of C major. The ratio between the frequencies of the notes E and C is:

$$2 \uparrow (4/12) = 1.25... \simeq 5/4$$

If we play all three notes together we get the simplest interesting set of frequency ratios possible with three notes. Such major chords are the most common in musical compositions. The ratios between the three notes are:

C (4 semitones) E (3 semitones) G (5 semitones) C



Many other chord structures exist, made up by combining thirds into fifths, sevenths, ninths and so on. You should also find out about the various standard chord sequences, such as that of the twelve bar blues, if you intend

to write your own music. For instance the chord sequence used in 'Roundabout' is G-C-F-D, which is known as a Round.

A Sequencer

The piece of software presented in this chapter corresponds to what is known by rock musicians as a sequencer. This is a 'black box' that, when connected to a synthesiser, can be programmed to remember any sequence of notes and play them back on demand.

The criteria used to design the program were that it should be interrupt driven (see Chapter 1), and that there should be an acceptable compromise between the amount of memory needed to store the music and, when the program was running, the amount by which the speed of the computer would be degraded. Initially the intent was to develop a music compiler – a program which, from some suitable input, would produce a machine code program directly controlling the SID chip by POKEing in values stored within the program. For example, an instruction such as:

```
LDA  #$0F
STA  $D418
```

would set the volume to 15. This is the most efficient method from the speed point of view, however it is not difficult to see that setting a single frequency will require ten bytes of memory, without having told SID to play the note!

Going to the other extreme, as there are nine octaves and hence 108 notes available from the SID chip, it should be possible to store the frequency, though not necessarily duration, data for each note in less than one byte. This is the approach taken here. Music is stored as a sequence of note numbers (see page 384 of the *Reference Guide* or the table given in the last chapter) with a separate 256 byte area for each voice. The note number is given by:

$$N=OCTAVE*16+NOTE$$

NOTE is 0 for C, 1 for C ♯ and so on. The actual high and low byte frequency values for each of the ninety-six possible notes are stored in a fourth 256 byte area of memory (see tables below). The formulae for these were given in the last chapter. Notice that we stick to Commodore's note numbering system, so that notes 13 to 15 do not exist, the second octave starts at note 16, the third at note 32, etc. The degradation in speed of the computer using this technique is negligible for most applications.

'Music' is an interrupt-driven music interpreter. The tempo of the music can be changed by altering the number of interrupts occurring before the interpreter is entered. For instance, if it is entered every fifth interrupt,

then the shortest musical note or rest possible is one tenth of a second long, as normal interrupts occur every fiftieth of a second. To play a double length note, the music table for the appropriate voice will have the normal number for that note in the usual position, but the next byte will contain a special value meaning either 'continue playing the note', or 'rest' (ungate the control register). As the largest note number is less than 128, the convention is adopted that bytes in a music table with the top bit set represent a special control function. Only two of these are actually implemented in the listing below (the other one is 'go back to start'), but it is very easy to add more if you have an assembler.

The 'Roundabout' program

The particular version of 'Music' we present here plays a two voice variation on the theme tune to the Magic Roundabout. If you do not possess an assembler, type in the program below. You should still read the rest of the chapter as it will explain how and where to put your coded music, how to change the settings of the SID chip to obtain other sounds, and, more importantly, how the program works. POKEing the SID chip directly will have no effect while this program is running. If you possess an assembler, type in the assembly language listing given later in this chapter.

```
1 REM"■=SAVE"@0:ROUNDABOUT.B",8
60000 FORI= 49152TO 49417:READA:POKEI,A:
NEXT
60010 DATA 76,49,234,174,124,192,232,142
,124,192,236,123,192,208,241
60020 DATA 169,0,141,124,192,76,128,192,
0,0,0,0,32,9,0,0,0,0,0
60030 DATA 32,9,0,0,0,0,0,0,0,0,120,169,
3,141,20,3,169,192,141
60040 DATA 21,3,88,96,120,169,49,141,20,
3,169,234,141,21,3,88,234
60050 DATA 169,0,162,0,157,0,212,232,224
,25,208,248,96,169,0,162,0
60060 DATA 157,0,212,232,224,25,208,248,
169,15,141,24,212,169,9,141
60070 DATA 5,212,169,0,141,6,212,169,9,1
41,12,212,169,0,141,13,212
60080 DATA 76,44,192,16,0,27,27,19,174,2
3,192,238,23,192,188,0,203
60090 DATA 162,0,32,173,192,174,30,192,2
38,30,192,188,0,204,162,7
```

```
60100 DATA 32,173,192,174,37,192,238,37,
192,188,0,205,162,14,32,173
60110 DATA 192,76,49,234,152,41,128,208,
42,185,0,202,157,1,212,185
60120 DATA 128,202,157,0,212,234,234,138
,168,105,4,141,127,192,185
60130 DATA 25,192,153,2,212,200,204,127,
192,208,244,188,27,192,200
60140 DATA 152,157,4,212,96,192,128,208,
7,189,27,192,157,4,212,96
60150 DATA 192,255,208,29,169,1,157,23,1
92,224,0,208,3,172,0,203,224
60160 DATA 7,208,3,172,0,204,224,14,208,
3,172,0,205,76,173,192,96
60170 DATA 64
61000 FORI= 51712TO 51968:READA:POKEI,A:
NEXT
61010 DATA 1,1,1,1,1,1,1,1,1,1,1,1,0,0,0
,0,2,2,2,2,2,2,2
61020 DATA 3,3,3,3,3,0,0,0,0,4,4,4,4,5,5
,5,6,6,7,7,7,0,0
61030 DATA 0,0,8,8,9,9,10,11,11,12,13,14
,14,15,0,0,0,0,16,17
61040 DATA 18,19,21,22,23,25,26,28,29,31
,0,0,0,0,33,35,37,39,42
61050 DATA 44,47,50,53,56,59,63,0,0,0,0,
67,71,75,79,84,89,94,100
61060 DATA 106,112,119,126,0,0,0,0,134,1
42,150,159,168,179,189,200
61070 DATA 212,225,238,253,0,0,0,0,12,28
,45,64,81,102,123,145,169
61080 DATA 195,221,250,0,0,0,0,24,56,90,
125,163,204,246,35,83,134
61090 DATA 187,244,0,0,0,0,48,112,180,25
1,71,152,237,71,167,12,119
61100 DATA 233,0,0,0,0,97,225,104,247,14
3,48,218,143,78,24,239,210
61110 DATA 0,0,0,0,195,195,209,239,31,96
,181,30,156,49,223,165,0
61120 DATA 0,0,0,135,134,162,223,62,193,
107,60,57,99,190,75,0,0
61130 DATA 0,0,15,12,69,191,125,131,214,
121,115,199,124,151,0,0
```

```
61140 DATA 0,0,30,24,139,126,250,6,172,2
43,230,143,248,46,0,0,0
61150 DATA 0,50
62000 FORI= 51968TO 52001:READA:POKEI,A:
NEXT
62010 DATA 50,55,50,55,50,55,50,55,52,55
,52,55,52,55,52,55,53,57
62020 DATA 53,57,53,57,53,57,50,57,50,57
,50,57,50,57,255,64
62100 FORI= 52224TO 52257:READA:POKEI,A:
NEXT
62110 DATA 71,128,71,71,66,128,66,128,68
,68,68,68,64,128,64,128,69
62120 DATA 128,69,69,64,64,64,64,70,70,7
0,70,66,128,66,66,255,64
62200 FORI= 52480TO 52482:READA:POKEI,A:
NEXT
62210 DATA 0,255,0
63000 SYS 49235
```

Save the program to tape or disk, then perform the following check. Change line 60000 to read:

```
B=0: FOR I=49152 TO 49417: READ A: B=B+A: NEXT: PRINT B
```

Perform the equivalent change to lines 61000, 62000, 62100 and 62200, and then delete line 63000. RUN the program. You should get the following values printed on your screen:

```
30896
17338
2035
2901
255
```

If your results agree with these 'checksums' then it is unlikely that there is anything wrong with your program. If they do not then check the corresponding sections of the program again, line by line, until you have got the data correct. Note that the first section, up to line 61000, is the machine code program. It is essential that this section is correct, or the program will not work. The second section is the frequency table of note values, with the high bytes in the first 128 bytes and the low bytes in the second 128 bytes. Note that the nonexistent note numbers (13 to 15 etc) have zero values. The

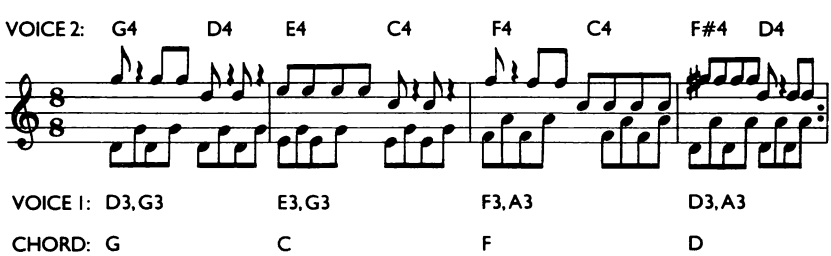
other three sections are the music tables for the three voices. Voice 3 just contains dummy entries.

When you have got the checksums correct, change back lines 60000, 61000, 62000, 62100, 62200 and 63000 to those given in the listing, and SAVE the correct version of the program (or reLOAD the original version, make any corrections and SAVE it again, if this is easier). It is unlikely that there is anything wrong with the program now. If it still does not run correctly you will just have to check it again.

Now comes the big moment. If you have not got the corrected version of 'Roundabout' in memory then LOAD it. Type RUN and after a moment you should hear the famous tune. Now type LIST. Before your eyes your program will list up the screen whilst the music is still playing. There are very few things you can do which will stop it. STOP will not but STOP/RESTORE will, as it sets the interrupt pointers back to normal. Even LOADing another program only slows the music down for a moment. The program has high line numbers so that you can attach it to the end of one of your own programs and then jump to it when you want the music to start. SYS49235 will start the interrupt routine and SYS49209 will switch it off again. Further on in this chapter we will give a couple of BASIC programs which can be RUN whilst 'Roundabout' is playing, and which will entirely change the character of the music.

Encoding the Music

Below is the music for our version of 'Magic Roundabout'. Note that there are two voices, one shown with the note tails up, the other with the tails down. There are 32 notes (including rests) for each voice. The symbol on the right-hand side means 'repeat from start'. The note names for Voice 1 are shown above the stave and for Voice 2, below the stave. At the bottom appear the chords used to construct the note sequences in each bar.



The tables for each Voice are constructed as follows for the first bar:

Voice 2			Voice 1		
Name	Decimal	Hex	Name	Decimal	Hex
G4	71	$47	D3	50	$32
—	128	$8C	G3	55	$37
G4	71	$47	D3	50	$32
G4	71	$47	G3	55	$37
D4	66	$42	D3	50	$32
—	128	$80	G3	55	$37
D4	66	$42	D3	50	$32
—	128	$80	G3	55	$37

Make sure you understand how to encode the rest of the music, taking particular note of the repeat symbol at the end. Hexadecimal dumps of the note table and coded music for 'Roundabout' are shown below. Note that all nine octaves appear in the note table, as on page 384 of the *Reference Manual*, so there is no need to change this for other pieces of music. The first 256 bytes contain the high frequency bytes and the second 256 bytes the low frequency bytes. Nonexistent notes (13 to 15 etc) are set to zero. The music tables appear one bar to a row. Voice 3 just contains the dummy 'repeat from start'. Remember that the three voices are interpreted entirely independently of one another so that each piece of music can be a different length, with different codas etc.

The Note Table

```
.:CA00 01 01 01 01 01 01 01 01
.:CA08 01 01 01 01 00 00 00 00
.:CA10 02 02 02 02 02 02 02 03
.:CA18 03 03 03 03 00 00 00 00
.:CA20 04 04 04 04 05 05 05 06
.:CA28 06 07 07 07 00 00 00 00
.:CA30 08 08 09 09 0A 0B 0B 0C
.:CA38 0D 0E 0E 0F 00 00 00 00
.:CA40 10 11 12 13 15 16 17 19
.:CA48 1A 1C 1D 1F 00 00 00 00
.:CA50 21 23 25 27 2A 2C 2F 32
.:CA58 35 38 3B 3F 00 00 00 00
.:CA60 43 47 4B 4F 54 59 5E 64
.:CA68 6A 70 77 7E 00 00 00 00
```

```
.:CA70 86 8E 96 9F A8 B3 BD C8
.:CA78 D4 E1 EE FD 00 00 00 00
.:CA80 0C 1C 2D 40 51 66 7B 91
.:CA88 A9 C3 DD FA 00 00 00 00
.:CA90 18 38 5A 7D A3 CC F6 23
.:CA98 53 86 BB F4 00 00 00 00
.:CAA0 30 70 B4 FB 47 98 ED 47
.:CAA8 A7 0C 77 E9 00 00 00 00
.:CAB0 61 E1 68 F7 8F 30 DA 8F
.:CAB8 4E 18 EF D2 00 00 00 00
.:CAC0 C3 C3 D1 EF 1F 60 B5 1E
.:CAC8 9C 31 DF A5 00 00 00 00
.:CAD0 87 86 A2 DF 3E C1 6B 3C
.:CAD8 39 63 BE 4B 00 00 00 00
.:CAE0 0F 0C 45 BF 7D 83 D6 79
.:CAE8 73 C7 7C 97 00 00 00 00
.:CAF0 1E 18 8B 7E FA 06 AC F3
.:CAF8 E6 8F F8 2E 00 00 00 00
```

Voice 1 Music Code

```
.:CB00 32 37 32 37 32 37 32 37
.:CB08 34 37 34 37 34 37 34 37
.:CB10 35 39 35 39 35 39 35 39
.:CB18 32 39 32 39 32 39 32 39
.:CB20 FF 40 FF FF FF FF FF FF
.:CB28 FF FF FF FF FF FF FF FF
.:CB30 FF FF FF FF FF FF FF FF
.:CB38 FF FF FF FF FF FF FF FF
.:CB40 FF FF FF FF FF FF FF FF
.:CB48 FF FF FF FF FF FF FF FF
.:CB50 FF FF FF FF FF FF FF FF
.:CB58 FF FF FF FF FF FF FF FF
.:CB60 FF FF FF FF FF FF FF FF
.:CB68 FF FF FF FF FF FF FF FF
.:CB70 FF FF FF FF FF FF FF FF
.:CB78 FF FF FF FF FF FF FF FF
```

Voice 2 Music Code

```
.:CC00 47 80 47 47 42 80 42 80
.:CC08 44 44 44 44 40 80 40 80
.:CC10 45 80 45 45 40 40 40 40
.:CC18 46 46 46 46 42 80 42 42
.:CC20 FF 40 FF FF FF FF FF FF
.:CC28 FF FF FF FF FF FF FF FF
.:CC30 FF FF FF FF FF FF FF FF
```

Voice 3 Music Code

```
.:CD00 00 FF 00 FF FF FF FF FF
.:CD08 FF FF FF FF FF FF FF FF
.:CD10 FF FF FF FF FF FF FF FF
.:CD18 FF FF FF FF FF FF FF FF
.:CD20 FF FF FF FF FF FF FF FF
```

The False SID chip

The technique used to handle the SID chip in this program is to hold a copy of it in memory locations $C017 to $C02B. Whenever it is required to play a note the entire FSID (False SID) chip area for the appropriate voice (FSID1, FSID2 or FSID3) is copied into the corresponding SID area, before the note is played, by the interpreter. Whenever it is required to change a value in the SID chip the corresponding register in the false SID chip is changed instead, which may be done by a foreground program. This method has the advantage that the FSID area can be PEEKed as well as POKEd to see what it contains.

FSID is not a true copy of the SID chip since registers 0 and 1 for each voice contain counters, which are used by the interpreter, rather than the frequency registers in SID. This is because the note frequency handling information is held in the note tables – it is the only parameter of SID which we would not want to change externally while the music is playing. Also the last eight registers in the SID chip do not have equivalents in the false SID chip. The last four are, in any case, read only.

When a note is played only five registers (2, 3, 4, 5 and 6) are copied from FSID to SID for any particular voice. We refer to all voice registers by these numbers, even though they only correspond to the SID chip register numbers for Voice 1.

In the last chapter of this book we present a similar technique for handling the VIC video chip, and discover that we can have as many false VIC chips as we like – and display them all on the screen at the same time!

False SID chips: locations

Decimal locations

REGISTERS	SID1	FSID1	FSID2	FSID3
0 Counter	—	49175	49182	49189
1 Subcounter	—	49176	49183	49190
2 Pulse (low byte)	54274	49177	49184	49191
3 Width (high nibble)	54275	49178	49185	49192
4 Waveform	54276	49179	49186	49193
5 Attack/decay	54277	49180	49187	49194
6 Sustain/release	54278	49181	49188	49195

Hexadecimal locations

REGISTERS	SID1	FSID1	FSID2	FSID3
0 Counter	—	$C017 (00)	$C01E (00)	$C025 (00)
1 Subcounter	—	$C018 (00)	$C01F (00)	$C026 (00)
2 Pulse (low byte)	$D402	$C019 (00)	$C020 (00)	$C027 (00)
3 Width (high nibble)	$D403	$C01A (00)	$C021 (00)	$C028 (00)
4 Waveform	$D404	$C01B (20)	$C022 (20)	$C029 (00)
5 Attack/decay	$D405	$C01C (09)	$C023 (09)	$C02A (00)
6 Sustain/release	$D406	$C01D (00)	$C024 (00)	$C02B (00)

The figures in brackets give the default settings for the 'Roundabout' program.

The 'Music' program

Monitor/assembler owners can type in the following program, making sure that they follow the line by line description of how it works. The structure of the program may seem rather strange, but there are good reasons for this. When this has been typed in, SAVE it on tape or disc and enter the hexadecimal dumps given above for the Note and Music tables. SAVE these and then, making sure you have all five sections (program, note table and three music tables) LOADed, type:

```
G C053
```

to start the routine running and:

```
G C039
```

to switch them off.

'Music' Program Listing

```
., C000  4C 31 EA JMP $EA31
., C003  AE 7C C0 LDX $C07C
., C006  E8       INX
., C007  8E 7C C0 STX $C07C
., C00A  EC 7B C0 CPX $C07B
., C00D  D0 F1    BNE $C000
., C00F  A9 00    LDA #$00
., C011  8D 7C C0 STA $C07C
., C014  4C 80 C0 JMP $C080
., C017  20 00 00 JSR $0000
., C01A  00       BRK
., C01B  20 09 00 JSR $0009
., C01E  20 00 00 JSR $0000
., C021  00       BRK
., C022  20 09 00 JSR $0009
., C025  01 00    ORA ($00,X)
., C027  00       BRK
., C028  00       BRK
., C029  00       BRK
., C02A  00       BRK
., C02B  00       BRK
., C02C  78       SEI
., C02D  A9 03    LDA #$03
., C02F  8D 14 03 STA $0314
., C032  A9 C0    LDA #$C0
., C034  8D 15 03 STA $0315
., C037  58       CLI
., C038  60       RTS
., C039  78       SEI
., C03A  A9 31    LDA #$31
., C03C  8D 14 03 STA $0314
., C03F  A9 EA    LDA #$EA
., C041  8D 15 03 STA $0315
., C044  58       CLI
., C045  EA       NOP
., C046  A9 00    LDA #$00
., C048  A2 00    LDX #$00
., C04A  9D 00 D4 STA $D400,X
., C04D  E8       INX
., C04E  E0 19    CPX #$19
., C050  D0 F8    BNE $C04A
., C052  60       RTS
```

```
., C053  A9 00     LDA #$00
., C055  A2 00     LDX #$00
., C057  9D 00 D4  STA $D400,X
., C05A  E8        INX
., C05B  E0 19     CPX #$19
., C05D  D0 F8     BNE $C057
., C05F  A9 0F     LDA #$0F
., C061  8D 18 D4  STA $D418
., C064  A9 09     LDA #$09
., C066  8D 05 D4  STA $D405
., C069  A9 00     LDA #$00
., C06B  8D 06 D4  STA $D406
., C06E  A9 09     LDA #$09
., C070  8D 0C D4  STA $D40C
., C073  A9 00     LDA #$00
., C075  8D 0D D4  STA $D40D
., C078  4C 2C C0  JMP $C02C
., C07B  10 03     BPL $C080
., C07D  1B        ???
., C07E  1B        ???
., C07F  13        ???
., C080  AE 17 C0  LDX $C017
., C083  EE 17 C0  INC $C017
., C086  BC 00 CB  LDY $CB00,X
., C089  A2 00     LDX #$00
., C08B  20 AD C0  JSR $C0AD
., C08E  AE 1E C0  LDX $C01E
., C091  EE 1E C0  INC $C01E
., C094  BC 00 CC  LDY $CC00,X
., C097  A2 07     LDX #$07
., C099  20 AD C0  JSR $C0AD
., C09C  AE 25 C0  LDX $C025
., C09F  EE 25 C0  INC $C025
., C0A2  BC 00 CD  LDY $CD00,X
., C0A5  A2 0E     LDX #$0E
., C0A7  20 AD C0  JSR $C0AD
., C0AA  4C 31 EA  JMP $EA31
., C0AD  98        TYA
., C0AE  29 80     AND #$80
., C0B0  D0 2A     BNE $C0DC
., C0B2  B9 00 CA  LDA $CA00,Y
., C0B5  9D 01 D4  STA $D401,X
., C0B8  B9 80 CA  LDA $CA80,Y
., C0BB  9D 00 D4  STA $D400,X
```

```
., C0BE  EA        NOP
., C0BF  EA        NOP
., C0C0  8A        TXA
., C0C1  A8        TAY
., C0C2  69 04     ADC #$04
., C0C4  8D 7F C0  STA $C07F
., C0C7  B9 19 C0  LDA $C019,Y
., C0CA  99 02 D4  STA $D402,Y
., C0CD  C8        INY
., C0CE  CC 7F C0  CPY $C07F
., C0D1  D0 F4     BNE $C0C7
., C0D3  BC 1B C0  LDY $C01B,X
., C0D6  C8        INY
., C0D7  98        TYA
., C0D8  9D 04 D4  STA $D404,X
., C0DB  60        RTS
., C0DC  C0 80     CPY #$80
., C0DE  D0 07     BNE $C0E7
., C0E0  BD 1B C0  LDA $C01B,X
., C0E3  9D 04 D4  STA $D404,X
., C0E6  60        RTS
., C0E7  C0 FF     CPY #$FF
., C0E9  D0 1D     BNE $C108
., C0EB  A9 01     LDA #$01
., C0ED  9D 17 C0  STA $C017,X
., C0F0  E0 00     CPX #$00
., C0F2  D0 03     BNE $C0F7
., C0F4  AC 00 CB  LDY $CB00
., C0F7  E0 07     CPX #$07
., C0F9  D0 03     BNE $C0FE
., C0FB  AC 00 CC  LDY $CC00
., C0FE  E0 0E     CPX #$0E
., C100  D0 03     BNE $C105
., C102  AC 00 CD  LDY $CD00
., C105  4C AD C0  JMP $C0AD
., C108  60        RTS
.
.
```

'Music' Program: Description

The program is in a number of sections, not necessarily in the order you might expect. (This is to make is easier to modify.) The routine to switch the interpreter into the interrupt routines is at $C053 to $C078 and $C02C to $C038. The routine to switch it back out again is at $C039 to $C052. Two RTSs at $C038

and $C052 must be changed to BRKs if you wish to run the program directly from the monitor rather than another program. The main interpreter is from $C000 to $C014 and $C080 to the end. $C080 to $C0AA gets the next note for each voice. $C0AD to $C0DB handles ordinary notes and $C0DC to the end handles special symbols.

C000–C014	Check to see if period between notes has expired
C000	Jump to normal interrupt service routines
C003	Load counter into **X**-register
C006/07	Increment and save counter
C00A	Compare counter to period
C00D	If not yet expired then exit
C00F/11	Reset counter to zero
C014	Jump to main interpreter
C017–C02B	False SID chips
C017	FSID1 counter
C018	FSID1 subcounter (not used in this version)
C019	FSID1 pulse width low byte
C01A	FSID1 pulse width high nybble
C01B	FSID1 control register
C01C	FSID1 attack/decay
C01D	FSID1 sustain/release
C01E/24	FSID2 as above
C025/2B	FSID3 as above
C02C–C038	Switch interpreter into interrupt routines. Note that entry is actually at $C053
C02C	Switch interrupts off whilst changing entry point
C02D/2F	Change low byte of entry point
C032/34	Change high byte of entry point
C037	Switch interrupts back on
C034	Return to calling program (change this to BRK if running from monitor)
C039–C052	Switch interpreter out of interrupt routines
C039/44	Change interrupt entry point back to normal
C039	Entry point for switching off interpreter

C045–C052 Clear SID chip

C046 Set Accumulator to zero
C048 Set **X**-register (counter) to zero
C04A Store zero in first register of SID chip+**X**
C04D Increment **X** (counter)
C04E If not end of SID chip branch to C04A
C052 Return to calling program (change this to BRK if running from monitor)

C053–C078 Initialisation routines

C053 Entry point for switching on interpreter
C055/5D Clear SID chip

C05F–C078 Set up SID chip

C05F/61 Set volume to maximum
C064/66 Set Voice 1 attack to 0, decay to 9
C069/6B Set Voice 1 sustain and release to zero
C06E/75 Set Voice 2 ADSR
C078 Jump to routine to change entry point

C080–C108 Main interpreter routine

C080–C08D Get notes for Voice 1

C080 Load FSID1 note counter into **X**-register
C083 Increment FSID1 counter
C086 Get note number from table. Put in **Y** (Indexed addressing: get number from $CB0X)
C089 Load **X** with zero. Value of **X** tells the program which voice we want:
Voice 1: **X**=0
Voice 2: **X**=7
Voice 3: **X**=E
C08B Go to note handling subroutine.

C08E–C099 Get note for Voice 2 (as above)

C09C–C0A7 Get note for Voice 3 (as above)

C0AA Exit to normal interrupt service routines

C0AD–C0DB Normal note handling subroutine

C0AD Check for special symbols: transfer note number to accumulator
C0AE Check if top bit is set

C0B0	Jump to symbol handling subroutines
C0B2	Get high byte of frequency from **Y**th position in note table
C0B5	POKE into **X**+1th register of SID chip
C0B8/BB	Transfer low byte of frequency from note table to SID chip
C0C0–C0D1	Move FSID chip into SID chip
C0C0/C1	**Y**=**A**=**X**=voice code
C0C2	Increment **A** register by four to get terminator for forthcoming loop
C0C4	Store in temporary variable area
C0C7–C0D1	Transfer bits 2 to 6 of false SID chip to real SID chip for appropriate voice
C0C7/CA	Transfer C019+**Y** to D402+**Y** (**Y**=0 to 4)
C0CD/CE	Increment counter and compare terminator
C0D1	Loop back to C0C7
C0D3–C0DB	Gate control register for appropriate Voice
C0D3	Fetch control register from FSID chip in **Y**
C0D6/D7	Set lowest bit and transfer to accumulator
C0D8	POKE into appropriate SID control register
C0DB	Return to Voice handling routine
C0DC–C0E6	Rest handling (release cycle)
C0DC/DE	If Y<>$80 go to next section
C0E0/E3	Transfer control register from FSID to SID (bit 0=0)
C0E6	Return to voice handling routine
C0E7–C108	Repeat from start. Set appropriate voice counter to 1
C0E7/E9	If Y<>$FF go to next section (NB: There isn't one!)
C0EB/ED	Set counter for appropriate Voice to 1
C0F0–C102	Get first note number from appropriate voice table into **Y**-register ready to return to note handling routine
C105	Return to note handling routine and play note 1 for appropriate Voice
C108	No more special symbols implemented. Return to calling program (ERROR!)

Try the checksums given for the BASIC program if you have any difficulty getting 'Roundabout' to work.

'Roundabout' – Important addresses

Program Areas

	START	FINISH
'MUSIC'	$C000	$C109
NOTES	$CA00	$CAFF
HIGH BYTES	$CA00	$CA7F
LOW BYTES	$CA80	$CAFF
DATA (maximum)		
Voice 1	$CB00	$CBFF
Voice 2	$CC00	$CCFF
Voice 3	$CD00	$CDFF
'Roundabout' Voice 1	$CB00	$CB21
Voice 2	$CCDD	$CB21
Voice 3	$CD00	$CD02

Interrupts:

INTERRUPT ON	$C053
INTERRUPT OFF	$C039

Variables and Constants:

TEMPO	$C07B
COUNTER	$C07C
SPARE	$C07D
SPARE	$C07E
TEMPORARY	$C07F

Special data values:

REST	$80
REPEAT	$FF

Sweeping effects

In this section we present a couple of programs to implement an alternative version of the dynamic effects we investigated in the last chapter. Dynamic effects were created by sweeping certain parameters in the SID chip up and down, in time with the notes that were playing, by transferring the contents of ENV3 (which was synchronised with the note) into the relevant SID register. In this section we use two small BASIC programs to achieve a similar effect except that this time the sweep is not synchronised with the SID chip

and just drifts up and down according to the length of a BASIC loop. Type in the following program and SAVE it.

'Pulse sweep' Program Listing

```
1 REM"■=SAVE"@0:PULSE",8
10 POKE49186,64
20 POKE49179,64
30 FORI=1TO16STEP0.1
35 PRINTINT(I)
40 POKE 49178,I:POKE 49185,I
50 NEXT
60 FORI=16TO1STEP-0.1
65 PRINTINT(I)
70 POKE 49178,I:POKE 49185,I
80 NEXT
90 GOTO30
```

Lines 10–20 set the pulse waveform for Voices 1 and 2. Lines 30–90 sweep the pulse width for Voices 1 and 2 up and down by varying the high nybble between 1 and 15.

Note that all POKEs are made to the false SID chip. LOAD all parts of 'Roundabout' and set it running. Next LOAD 'Pulse sweep' and run it. You will be able to hear the pulse width of each voice varying between a thin reedy sound and a full square wave.

The next program sweeps the bandpass filter frequency up and down the frequency spectrum producing a 'wah' type sweep on Voice 1.

'Filter sweep' program Listing

```
1 REM"■=SAVE"@0:FILTER",8
10 POKE54296,15+32:POKE54295,15*16+1
30 FORI=0TO255STEP10
35 PRINTINT(I)
40 POKE 54294,I
50 NEXT
60 FORI=255TO0STEP-10
65 PRINTINT(I)
70 POKE 54294,I
80 NEXT
90 GOTO30
```

Line 10 sets volume to 15 (filter to bandpass) and resonance to 15 (filter Voice 1 only). Lines 30–90 sweep the contents of high byte of filter frequency up and down. Again, RUN this program at the same time as 'Roundabout' is playing.

Changing the loop terminator and step parameters in these two programs will change the sound and feel of the music. As an exercise, try implementing a swept flanging effect by sweeping the frequency of Voice 3 up and down with either ring modulation and triangle waveform or synchronisation set.

Development

For simplicity the version of 'Music' presented here has only two special symbols, rest and repeat. Try implementing the following developments:

1. Extended note length (i.e. the same as rest but do not ungate register: do nothing).

2. Codas: a coda is normally a short passage of music, within the main body, which is repeated a number of times.

3. Use Voice 3 of SID to produce a percussion backing for 'Roundabout.' Set Voice 3 to noise, with short attack and decay rates and a low sustain level with a long release.

4. Switch special effects on and off during the piece of music. Choose special symbols for changing waveforms, ring modulation, synchronisation, etc.

5. Choose special symbols for changing parameters, ADSRs, pulse width, etc. during the piece of music. Reserve the next couple of bytes after the symbol for the value of the parameter.

6. Implement dynamic effects. To do this you will have to change the parameter values every couple of interrupts or so. Thus lines C000 – C014 will have to be modified.

7. Work out a way of increasing the number of notes per voice beyond 256, which is the current maximum.

Note that the technique of using a table of codes along with a very fast interrupt driven interpreter can not only be used to drive the SID chip, but could also drive, or sequence, an interactive animated cartoon, game, educational program or whatever.

I suggest the following conventions for special symbols:

128	$80	rest (i.e. enter release cycle)
129	$81	tied note (i.e. continue playing note)
. . .		other time symbols
	$90	
. . .		special effects on and off
	$A0	
. . .		change parameters (remember to reserve a couple of bytes for the new parameter value)
	$B0	
. . .		dynamic effects on and off
	$D0	
	$E0	remember this position (coda)
	$E1	remember this position (coda)
. . .		
		($E0 to $EE should be followed by a byte for the number of codas.)
. . .		
	$EF	
	$F0	return to $E0
	$F1	return to $E1
. . .		
255	$FF	repeat from start

The code to test these new symbols should just be tagged on to the program. For instance, to implement extended note lengths, just change line C108 to read:

```
C108    CPY #$81
C10A    BNE . . .
        . . .
        RTS
```

It is not suggested that you write a program that will do all the above, though this is possible, but re-design your program for each musical application. Repetitive works are always easiest to implement, provided that the codas are not too convoluted. Long guitar solos must be coded in full, bar by bar, whereas heavy complex disco or jazz-funk rhythms can be reduced down to a few bars, repeated over and over. Do not restrict Voices 1, 2 and 3 to their associated Voice tables, but allow them to swap over. Remember that the interpreter can always receive messages from the foreground program and vice versa. Code up a major piece of music, or try allowing the interpreter to play random music! Tell it that the note or music tables start at C000 and it will play itself! (This sounds quite good.) Above all else, experiment!

4 Graphics

This chapter describes memory management for graphical work, gives a register by register treatment of the VIC II chip, introduces the five available graphics modes and explains how to make the best use of bit-mapped graphics on your computer.

Memory Management

The VIC II chip of the 64 sees forty-seven registers in RAM from $D000 (53248) to $D02E (53294). Most locations are read *and* write (unlike the SID registers). These control eight real sprites and allow five graphics modes. It also sees a further 16K block of memory, normally from $0000 to $3FFF (16383) containing the BASIC screen, sprite data and the bit-mapped (high resolution) screen plus its associated colour map. In this book we will normally relocate this block to sit in the area $4000 (16384) to $7FFF (32767). VIC also takes information from the 'colour nybble' area, from $D800 (55296) to $DC00 (56319), and the character ROM, from $D000 (53248) to $DFFF (57343). The colour nybbles control the character colours of the text screen, and scroll with it. The character ROM contains the shapes of the text and graphics characters.

We have already discussed the SID chip. There are two other special integrated circuits in the 64. These are the Complex Interface Adapters CIA1 and CIA2. They control paddles, joysticks, serial (RS232C) and parallel (IEEE488) interfaces and the computer timers, including the real-time clock (accessed by printing TI$ in BASIC). They sit in the same 4K Input-Output area of RAM as SID and VIC, as shown below:

Input-Output Area Addresses

	Hex	Decimal
VIC CHIP	D000–D02E	53248–53294
VIC IMAGES	D040–D3FF	
SID CHIP	D400–D41C	54272–54300
SID IMAGES	D420–D7FF	
COLOUR NYBBLES	D800–DBFF	55296–56319
CIA1	DC00–DC0F	56320–56335
CIA1 IMAGES	DC10–DCFF	
CIA2	DD00–DD0F	56576–56591
CIA2 IMAGES	DD10–DDFF	
RESERVED	DE00–DEFF	
RESERVED	DF00–DFFF	

The 'images' are copies of the corresponding chips. Changing a value in the VIC chip will change the corresponding value in every VIC image simultaneously. I suspect that the SID images may be used by Commodore's 'soon to be released' musical keyboard.

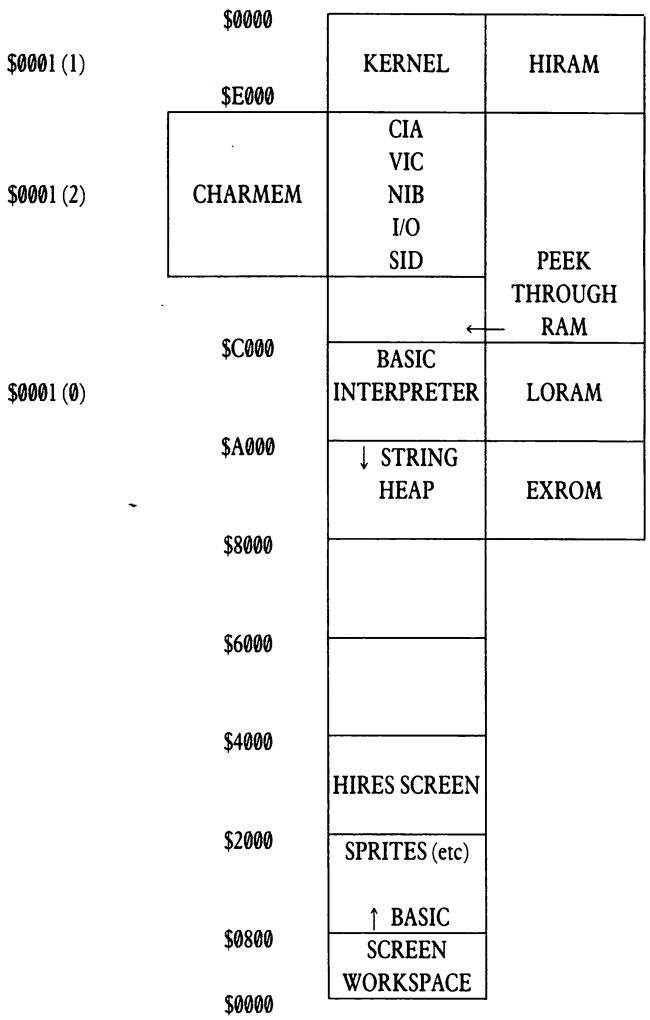
Sharp eyed readers may have noticed that the character ROM and the Input-Output Area occupy the same 4K of memory address space. (To make things worse there is also 4K of RAM underneath.) This does not bother the computer – it knows where everything is – but there must be a way for the user to know whether he is accessing ROM, RAM or the I/O chips. This control is provided by setting bit 2 of the 64's Input-Output register (at location $0001), to 0 to switch the ROM in, and 1 to switch it out.

There are two other 8K blocks of memory that can be treated like this. The 64 operating system 'Kernel' normally occupies locations $E000 (57344) to $FFFF (65535). Setting bit 1 of $0001 to 0 will exchange it for an 8K block of RAM. *Kernel cannot be used while the computer is in this state.* Similarly the BASIC interpreter normally occupies locations $A000 (40960) to $BFFF (49151). Setting bit 0 of $0001 to 0 will exchange it for another 8K block of RAM, *but BASIC cannot be used while the computer is in this state* (which is fine for machine code programs or subroutines). Try POKE 1,0 in BASIC. Do not do this unless you want to switch your computer off!

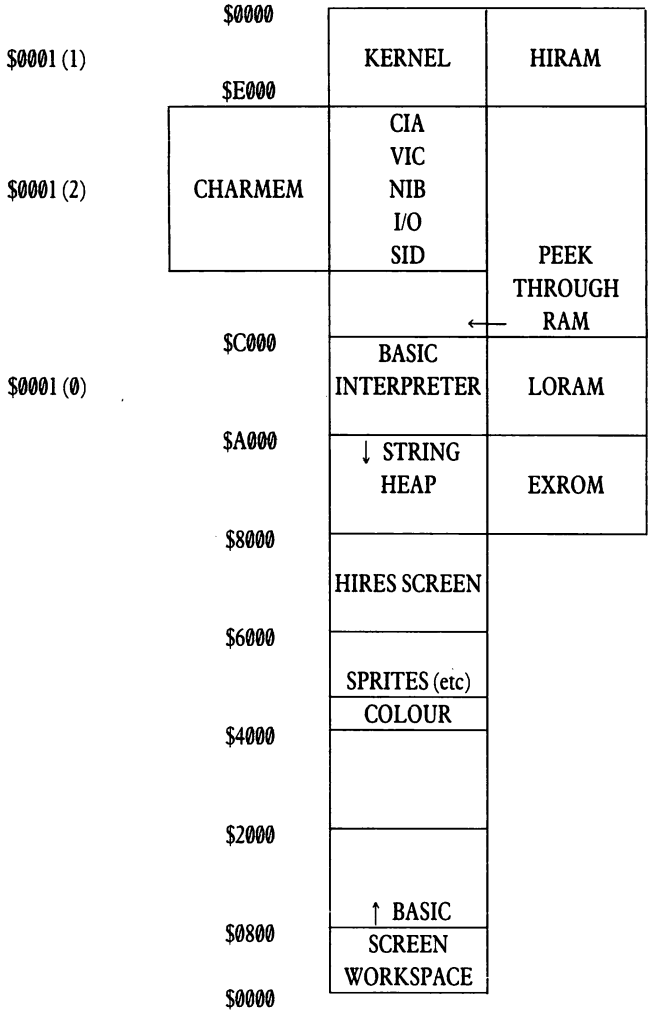
As mentioned above, VIC normally sees (amongst other things) the first 16K of RAM for its high resolution screen area. This is very inconvenient for graphics work as it means that the BASIC screen and the graphics colour map occupy the same area of memory, and that we cannot have a BASIC program more than 6K long without overwriting the high resolution screen – and this

figure is even worse if we want to use sprites as well. The solution is to point VIC at one of the other three 16K blocks of RAM in the computer. In this book we use the second block from $4000 to $7FFF. This means that we do not have to mess around switching Kernel or BASIC in and out to access the screen RAM, so we can do everything in BASIC, but has the disadvantage that we are limited to 14K of program. Also, since BASIC stores its strings in a 'heap' from $A000 downwards it is necessary to call FRE(0) occasionally to prevent the heap overwriting the screen area. For efficiency, Screen Graphics 64 stores the high resolution screen under the BASIC interpreter and Simons' BASIC stores it under Kernel. The VIC chip always sees the high resolution screen area (when told to), but the machine code routines in Screen Graphics and Simons' BASIC must switch the RAM in or out if they need to access the bit mapped screen. Memory maps of these four configurations are shown in the diagrams that follow:

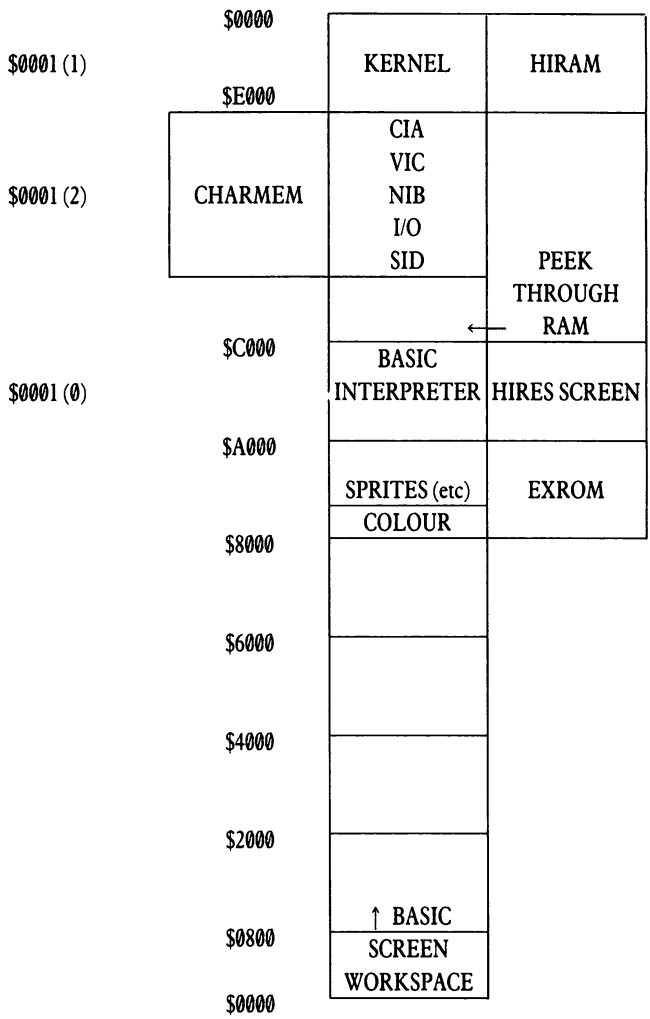
1. Default Memory Organisation



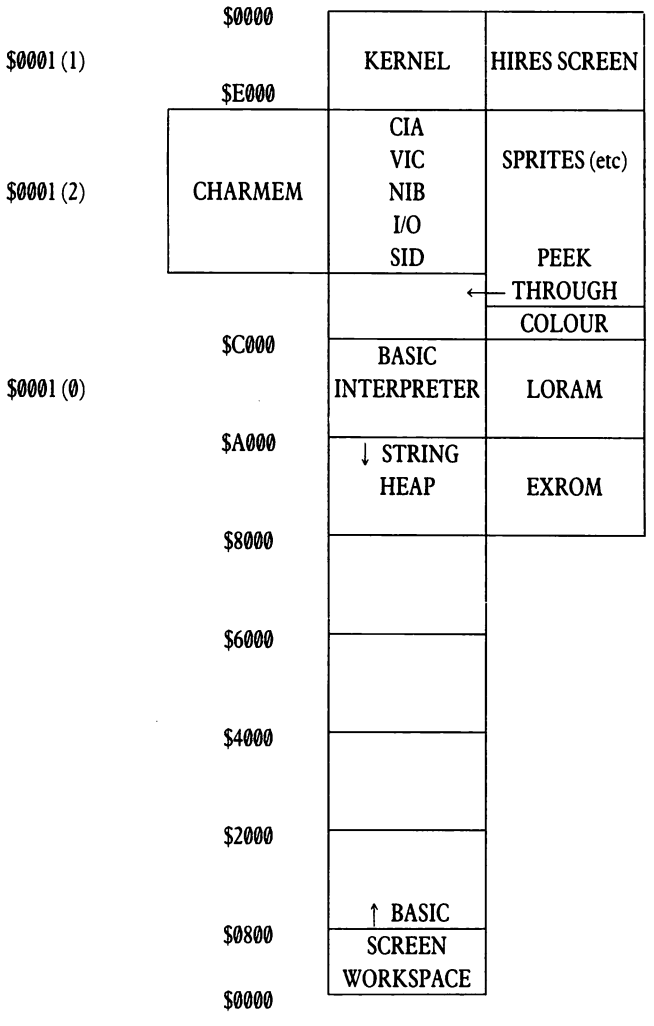
2. Memory Organisation for this Book



3. Memory Organisation used by Screen Graphics 64 and Ultrabasic



4. Memory Organisation used by Simons' BASIC



The reconfiguration we use (number 2 above) is controlled by the first two bits of location $DD00 (56576), the first register of CIA2, as follows:

PEEK (56576) AND	VIC BANK	BANK NUMBER
3	$0000–$3FFF	0
2	$4000–$7FFF	1
1	$8000–$BFFF	2
0	$C000–$FFFF	3

The BASIC screen area can also be moved up and down within VIC's 16K memory bank in 1K steps, by POKEing the upper nybble of location $D018

(53272) in the VIC chip (0 for the first 1K block, 1 for the second, etc). This is important as in bank 0 the first 1K block is the system work area. Thus, as the default bank is $0000–$3FFF, the default screen area uses $0400 to $04FB. To switch VIC to bank 1 and move the screen area into the first 1K block of this we must enter:

```
POKE 56578, PEEK (56578) OR 3
POKE 56576, (PEEK (56576) AND 252) OR 2
POKE 53272, PEEK (53272) AND 15
```

To simplify expressions such as these in the programs given in the next two chapters we simply evaluate the expressions on the right hand side of the statements, given the default (normal) values of the registers, and include these in the POKE commands.

In a similar way, bits 1–3 of location $D018 (53272) govern the location of character memory within VIC's 16K block. The VIC chip is fooled into seeing 2K of character ROM/RAM which can be moved up and down the 16K block in 2K steps, even though the character ROM is actually at locations $D000–$DFFF. For most purposes you do not have to know this, as the 2K RAM above the ROM image can be used normally. The default locations for the ROM image are $1000 (4096) to $17FF (1643). Note that the 64's character set is not available to the VIC chip from banks 1 or 3. A program using these must switch VIC to one of the other banks to display text.

The VIC chip

The VIC II chip registers can be split into three groups. Registers 0 to 15 control the X and Y positions of the eight real sprites. However, since there are more than 320 points across the screen and the **X**-registers can only store numbers between 0 and 255 it is necessary to use an extra bit for each sprite to define whether it is on the left or right side of the screen. Bit 7 of register 16 gives the most significant bit (MSB) of sprite 7 and bit 0 the MSB of sprite 0. Thus the X position of sprite N is given by:

$$X=256*MSB(N)+SX(N)$$

where MSB (N) is the Nth bit of register 16 and SX(N) is the **X**-register for sprite N.

Register 17 controls the following attributes:

Bit	Name	Effect
7	RC8	MSB of raster position
6	ECM	Extended colour mode
5	BMM	Bit mapped mode
4	BLNK	Blank screen
3	RSEL	Number of rows (0=24 rows)

2	YSCL2	Smooth scroll in Y-direction
1	YSCL1	
0	YSCL0	

Register 18 is the rest of the raster position register, RC7 to RC0 (see Chapter 8). This nine bit number gives the current position of the television raster scan. If POKEd with a raster value it will cause an interrupt next time the raster reaches that value. Registers 19 and 20 give the X and Y light pen positions. Bits 7 to 0 of register 21 causes sprites 7 to 0 to appear when set.

Register 22 controls the following attributes:

Bit	Name	Effect
7	NC	Not used
6	NC	Not used
5	RST	Switch VIC chip off (always set to 0!)
4	MCM	Multicolour mode
3	CSEL	Number of columns (0=38 cols)
2	XSCL2	Smooth scroll in X direction
1	XSCL1	
0	XSCL0	

Bits 7 to 0 of register 29 set sprites 7 to 0 to double width. Bits 7 to 0 of register 23 expand the sprites in the Y direction.

Register 24 controls character memory position, with bits 7–4 controlling the video matrix base address, and bits 3–1 controlling the base address of the character dot data.

Register 25 and 26 control interrupt requests as follows. Register 25 is the interrupt flag register. If a bit is set then an interrupt has occurred for the associated attribute. Register 26 controls the interrupt request masks. If a bit is set then interrupts for the associated attribute are enabled.

Bit	Register 25	Register 26	Effect
7	IRQ	NC	Set on any enabled VIC interrupt
6	NC	NC	Not used
5	NC	NC	Not used
4	NC	NC	Not used
3	LPIRQ	MLPI	Light pen interrupt
2	ISSC	MISSC	Sprite to sprite collisions interrupt
1	ISBC	MISBC	Sprite to foreground collisions
0	RIRQ	MRIRQ	Raster compare interrupt

Register 27 controls the sprite foreground priority. If the Nth bit is set then sprite N will be seen in front of the foreground. Bits 7 to 0 of register 28 set multicolour mode for sprites 7 to 0. Bits 7 to 0 of registers 30 and 31 are set if sprites 7 to 0 are involved in a collision with another sprite, or with the screen foreground, respectively. The third section, registers 32 to 46, controls screen colours as in the following master table.

THE VIC CHIP

Register	Address		Function		Defaults	
	hex	decimal			decimal	hex
0	D000	53248	Sprite 0 X position		0	0
1	D001	53249	Sprite 0 Y position		0	0
2	D002	53250	Sprite 1 X position		0	0
3	D003	53251	Sprite 1 Y position		0	0
4	D004	53252	Sprite 2 X position		0	0
5	D005	53253	Sprite 2 Y position		0	0
6	D006	53254	Sprite 3 X position		0	0
7	D007	53255	Sprite 3 Y position		0	0
8	D008	53256	Sprite 4 X position		0	0
9	D009	53257	Sprite 4 Y position		0	0
10	D00A	53258	Sprite 5 X position		0	0
11	D00B	53259	Sprite 5 Y position		0	0
12	D00C	53260	Sprite 6 X position		0	0
13	D00D	53261	Sprite 6 Y position		0	0
14	D00E	53262	Sprite 7 X position		0	0
15	D00F	53263	Sprite 7 Y position		0	0
16	D010	53264	MSB of X co-ordinates		0	0
17	D011	53265	Mode (Y scroll)		27	1B
18	D012	53266	Raster register		—	—
19	D013	53267	Light pen X co-ordinate		—	—
20	D014	53268	Light pen Y co-ordinate		—	—
21	D015	53269	Sprite appear		0	0
22	D016	53270	Mode (X scroll)		200	C8
23	D017	53271	Sprite expand (Y)		0	0
24	D018	53272	Character memory position		21	15
25	D019	53273	Interrupt requests		121	79
26	D01A	53274	Interrupt enable register		240	F0
27	D01B	53275	Foreground-sprite priority		0	0
28	D01C	53276	Multicolour sprite select		0	0
29	D01D	53277	Sprite expand (X)		0	0
30	D01E	53278	Sprite-sprite collisions		0	0
31	D01F	53279	Sprite-background collisions		0	0
32	D020	53280	Border colour	(14)	254	FE
33	D021	53281	Background colour 0	(6)	246	F6
34	D022	53282	Background colour 1	(1)	241	F1
35	D023	53283	Background colour 2	(2)	242	F2
36	D024	53284	Background colour 3	(3)	243	F3
37	D025	53285	Sprite multicolour 0	(4)	244	F4
38	D026	53286	Sprite multicolour 1	(0)	240	F0
39	D027	53287	Sprite 0 colour	(1)	241	F1
40	D028	53288	Sprite 1 colour	(2)	242	F2
41	D029	53289	Sprite 2 colour	(3)	243	F3
42	D02A	53290	Sprite 3 colour	(4)	244	F4
43	D02B	53291	Sprite 4 colour	(5)	245	F5
44	D02C	53292	Sprite 5 colour	(6)	246	F6
45	D02D	53293	Sprite 6 colour	(7)	247	F7
46	D02E	53294	Sprite 7 colour	(12)	252	FC

Graphics Modes

The five graphics modes are:

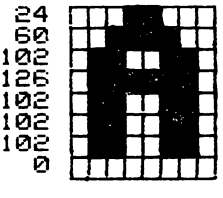
1. Normal text mode with twenty-five rows of forty characters on the screen, each character being made up of eight rows of eight pixels. The background colour of each character is fixed for the whole screen, using register $D021 (53281) but the character (i.e. the foreground) may be any one of sixteen colours. The foreground colour data is stored in the 'colour nybble' area, located from $D800 (55296) to $DBFF (56319). Only the first 1000 bytes of this block are used, and only the lower nybble is used for storing colour data. In this mode, all of the nybble is used for colour, and so there is a choice of 2 ↑ 4=16 colours.

The actual character shapes are stored in the character ROM, or in the underlying RAM if user-defined characters are being used. The screen area from 1024 onwards (again, only the first 40*25=1000 locations) contains a set of pointers to shape which should be shown in each screen position.

The first byte of screen memory is at location 1024, the top left corner of the TV screen. If you type PRINT PEEK(1024) <RETURN> then the number returned is the pointer to the position of the character within the character ROM. If location 1024 contains a blank space then PEEK(1024) will return the value 32. The 32nd character in the character ROM is a space. Similarly, the letter R is number 1 in the ROM and PEEK(1024) after putting an R in the top left corner of the screen will thus return the value 1.

A single byte can hold any number between 0 and 255 so the maximum number of different characters that can be shown on the screen simultaneously is 256. There are two complete character sets in the 64, and switching from one to the other is simple. Press the SHIFT and CBM keys simultaneously and you will see the change!

As was mentioned at the beginning of this chapter, each character square on the screen is made up of eight rows of eight bits, where each bit may be either a 1 (the pixel is on) or a 0 (the pixel is off). Each character image is stored in the character ROM as a series of eight bytes, with each byte representing the pixel pattern of a row in the screen character. The letter A, for instance, is stored as 24,60,102,126,102,102,102,0 as here:



In normal text mode the computer will interpret this data from the character ROM directly to set the necessary bits to form each character on the screen.

2. Multicolour text mode, with twenty-five rows of forty characters. This is turned on by setting bit 4 of $D016 (53270) to 1. In this mode each character square can contain four colours, but there is a loss of horizontal resolution. Each character now consists of eight rows of four pixel-pairs and the colour of each of these pixel-pairs is dependent upon the associated bit-pair pattern. The character is still the same size, but each dot on the screen is now twice as wide.

The colour for each pixel-pair is defined as follows:

Bit Pair	Level	Colour Defined By:	Location
00	background	background 0 colour	53281 ($D021)
01	background	background 1 colour	53282 ($D022)
10	foreground	foreground 0 colour	53283 ($D023)
11	foreground	lower 3 bits of	colour nybbles

So, to continue with the example of letter A, in multicolour mode it would now take colours thus:

B0	B1	F0	B0
B0	F1	F1	B0
B1	F0	B1	F0
B1	F1	F1	F0
B1	F0	B1	F0
B1	F0	B1	F0
B1	F0	B1	F0
B0	B0	B0	B0

It is, of course, unrecognisable as a letter A. You should notice that colours B0, B1 and F0 are the same for every character across the entire screen, but that colour F1 can be different in each square. In multicolour mode only the lowest three bits of each colour nybble are used for colour definition and therefore you have a choice of 2 ↑ 3=8 colours (the colours on the number keys). The highest bit is used to determine whether or not that character position is *actually* in multicolour mode, because it is possible to have multicoloured and normal characters on the screen at the same time. If the highest bit of a nybble (4 bits) is set (1) then that character will be in multicolour mode, and if it is unset (0), then the character will be in normal (high resolution) mode.

Although this may seem a complicated process it is in fact very simple to switch multicolour mode on or off using the colour controls on the keyboard, either directly or under program control. High resolution (normal) characters can be printed in any of eight colours using the <CTRL> key and keys 1–8, with the chosen colour shown on the front of the key. By

using the <CBM> key together with keys 0–8 then the same colour will be selected but all following characters will be in multicolour mode. The selected colour is colour F1 in the table above.

The screen can be thought of as being divided into two 'layers', foreground and background. Sprites may pass either in front of the foreground, or between the foreground and the background, depending upon the value of the associated bit in the Foreground-Sprite Priority register in the VIC chip. Note that when in multicolour text mode the use of foreground priority (i.e. sprites passing behind a foreground colour) may produce unreal results with the sprite apparently moving in front of part of a multicoloured character *and* behind parts of the same character. This effect is the result of the computer interpreting two of the four colours as background colours and the other two as foreground colours.

3. Extended background colour text mode, with twenty-five rows of forty characters. Extended background colour text mode is turned on by setting bit 6 of location $D011 (53265) to 1. In this mode only the lowest 6 bits of each byte of screen memory are used as a pointer into character ROM. The two highest bits are used to store the background colour of each character square.

Bit 6 & 7 Bit Pair	Screen Byte Range	Colour Defined By	Colour Number
00	0– 63	53281 ($D021)	B1
01	64–127	53282 ($D022)	B2
10	128–191	53283 ($D023)	B3
11	192–255	53284 ($D024)	B4

Since only the lowest 6 bits of screen memory are used as pointers to character memory, only the first 2 ↑ 6=64 characters can be shown on the screen in this mode. However, the various background colours can be accessed very easily by using combinations of reversed and/or SHIFTed characters. Printing a REVERSE character will in fact give the same character, but with a background colour determined by the contents of location 53282 (B2). Similarly, a SHIFTed character will give the same character with background colour B3 and a RVSed and SHIFTed character will have background colour B4.

It should be noted that multicolour mode and extended background colour mode cannot both be enabled at the same time. Both modes use the same registers for accessing colour data but in different ways. When using both modes at the same time the computer may be called upon to act in two contradictory ways in response to a single piece of data. Computers, of course, dislike ambiguity, and to cover its confusion the 64 will present you with a totally blank and black screen!

4. High resolution bit mapped graphics mode, with 200 rows of 320 pixels.

Only two colours are allowed in each 8*8 pixel character square. Thus a screen of high resolution graphics takes 320*200/8=8000 bytes of RAM. The colour area is within the VIC's 16K block and takes 40*25=1000 bytes. The colour nybbles are not used. This allows each character square to have an independent foreground and background colour, and stops the colours scrolling with the text screen. Bit mapped graphics is turned on by setting bit 5 of location $D011 (53265) to 1.

Each pixel on the screen corresponds to a bit in memory. If the bit is set to 1, that pixel is set to the foreground colour, for that character square, defined by the upper nybble of the corresponding byte in colour RAM, and if set to 0, the pixel is set to the background colour for that character square, defined by the value of the lower nybble of the corresponding byte in colour RAM. The screen is mapped to memory as follows. The first 320 bytes of the (just under) 8K of screen area defines what will appear on the first (top) row of 8*8 pixel blocks (character squares) on the screen. The second 320 bytes defines the second row of character squares, and so on. Each row is mapped such that the first character square (8*8 pixel block) in the row is defined by the first eight bytes in the associated 320 byte block, the second character square is defined by the second eight bytes, and so on. For each character square the first byte defines the first row of eight pixels, and the second byte, the second row, exactly as with character memory. Thus, in high resolution mode, every character square can display a different character and we are not limited to 256 as in text mode.

Colour RAM is mapped to the screen as follows. Each byte defines the foreground and background colour of a character square, as described above. The first forty bytes of colour RAM define the colours of the first row of character squares on the screen, the second forty bytes the colours of the second row, and so on.

5. Multicolour bit mapped mode, with 200 rows of 160 pixels and four colours in each character square. One of the colours (background colour) is fixed over the whole screen, and the others depend on the character square. Multicolour bit mapped mode is turned on by setting both high resolution bit mapped mode and multicolour mode. The screen map is identical to that for high resolution bit mapped mode except that each pixel is associated with two bits in memory, the byte being split into pairs of bits in the same way as with the multicolour mode. The colour of the pixel and its level are defined as follows:

Bit Pair	Level	Colour Defined By:	Location
00	background	Background colour	53281 ($D021)
01	background	Upper 4 bits of	Colour RAM
10	foreground	Lower 4 bits of	Colour RAM
11	foreground	4 bits of	Colour nybble

This is a very powerful mode, but it should be noted that foreground colour nybbles will still scroll with the text screen.

Modes 4 and 5 will be treated in depth in the following two chapters. We'll now cover in the rest of this chapter some topics inadequately dealt with in the *Reference Manual*. We do not cover here smooth scrolling, use of joysticks and paddles, or collision detection, as the first two are adequately covered in the *Reference Manual*, and the last is just a variation on the techniques used in Chapter 8.

Bit mapped text modes

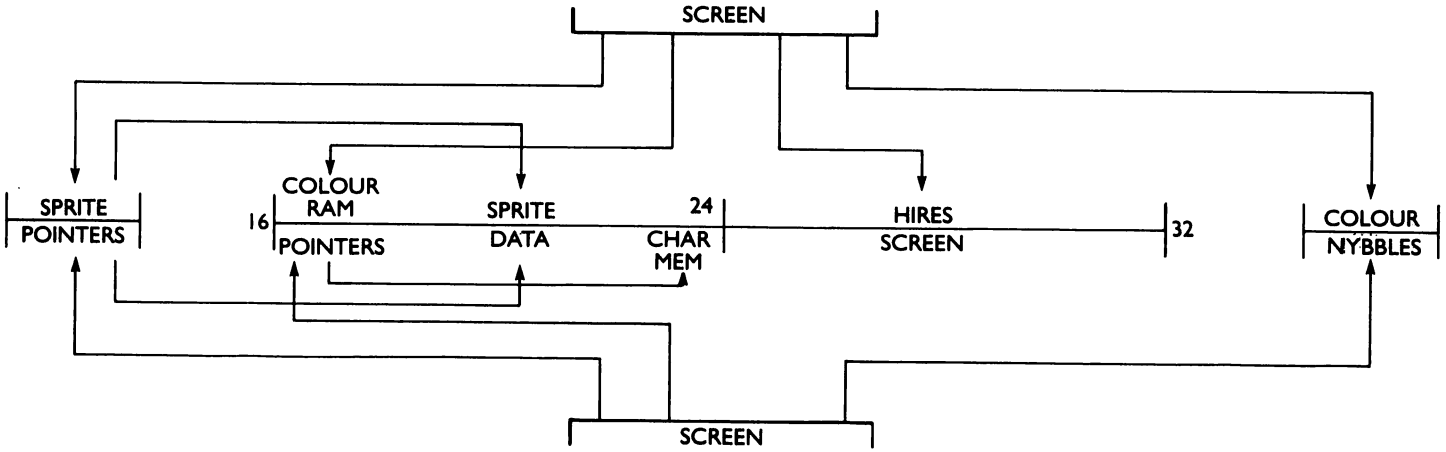
It should be noted that the text modes are far more powerful for animation purposes than the bit mapped graphics modes. This is because they use a set of pointers from each character square to the character memory. If every character square is set to point to the same area in character RAM, and the data in that area is then changed, the whole screen display will change simultaneously.

Unfortunately, only 256 characters can be displayed on the screen at once, which means that only one quarter of the thousand-character text screen can be bit mapped using this technique. However, if the screen is split into four equal sections, using raster interrupt techniques, and character RAM is banked between four corresponding 2K blocks on each interrupt, we can then create three new bit mapped text modes! These powerful modes map one bit in memory to one pixel on the screen, but still use the screen pointers to indirectly map the character squares, allowing rapid global animation.

All the techniques necessary for implementing bit mapped text modes are given in this book, but the actual implementation is left as an exercise for the reader.

DISPLAY MODES

BIT MAPPED MODE



TEXT

Optimising high resolution colour graphics

The next two chapters cover the implementation of two highly sophisticated 'Etch-a-Sketch' type screen editors, one for high resolution drawing, and the other for multicolour mode. The simplest method of transferring a detailed colour picture onto the 64's screen is to take a piece of tracing paper, the size of your picture, and divide it up into a rectangular grid, with forty squares in the X-direction and twenty-five in the Y-direction. If you want the X and Y scales of the final picture to be equal (no distortion) then you must take account of this when defining the grid.

Overlay the piece of tracing paper on top of the picture. Now you must go through each of the thousand squares in your grid and choose the two colours which you are going to allow within this square of the high resolution bit mapped mode – which, of course, should be the two most dominant colours in the corresponding character square in the original picture. Finally, remembering that there is an 8*8 array in each square, and that each pixel can be set either to the foreground or background colour for that square, the entire picture can be transferred, square by square, on to the high resolution screen.

As an exercise, when you have completed the programs in the next two chapters, you could add an extra routine to the two screen editors which will display a blown up representation of the current character square using a sprite, with any changes to the square applied to the sprite simultaneously. This will make it far easier to perform a square by square transfer.

If you are designing your own background pictures for games, or whatever, then the above technique is probably the best, as freehand drawing on the screen using an artist program causes definite difficulties with character square graphics.

You may have noticed that there is a considerable degree of redundancy in the way the 64 stores its high resolution screen, unless you are making use of the sprite foreground and background layers. If the background and foreground colours for a particular character square are swapped, and each byte of the bit mapped screen for that square is inverted (i.e. the 0s and 1s are swapped), then the picture remains unchanged. This inefficiency is a minor price to pay for such a powerful graphics mode, and has the advantage that more than one approach is available for picture design. Thus:

1. Make the dominant colour in a character square the background colour, and the second most dominant colour the foreground. This has definite advantages when saving the screen to magnetic media, as the high resolution screen data can be compressed into a very efficient form. A good technique here is to first draw the entire screen in low resolution,

setting the bit mapped screen entirely blank and then simply changing the background colours. Then round off the 'staircase' effect using the foreground colours. This technique is quite good for freehand drawing.

2. Make the background, the background colour, and the foreground, the foreground colour. This is essential if you wish to pass sprites behind any part of the foreground and get sensible results. That is, if you are drawing a brown tree in a green field, use 1's for the tree and 0's for the field. Set the foreground colour to brown and the background colour to green. This technique causes problems if there are more than two layers required, but is essential if you want to be able to pass a sprite behind the tree and in front of the field. If more than two screen layers are required then static sprites can be used for some of the layers, or the priorities of the sprites may be changed as they move across the screen.

3. Alternate foreground and background colours between adjacent areas, e.g. for a rainbow, the primary colours could be drawn in foreground colours and the secondary colours in background colours:

R red

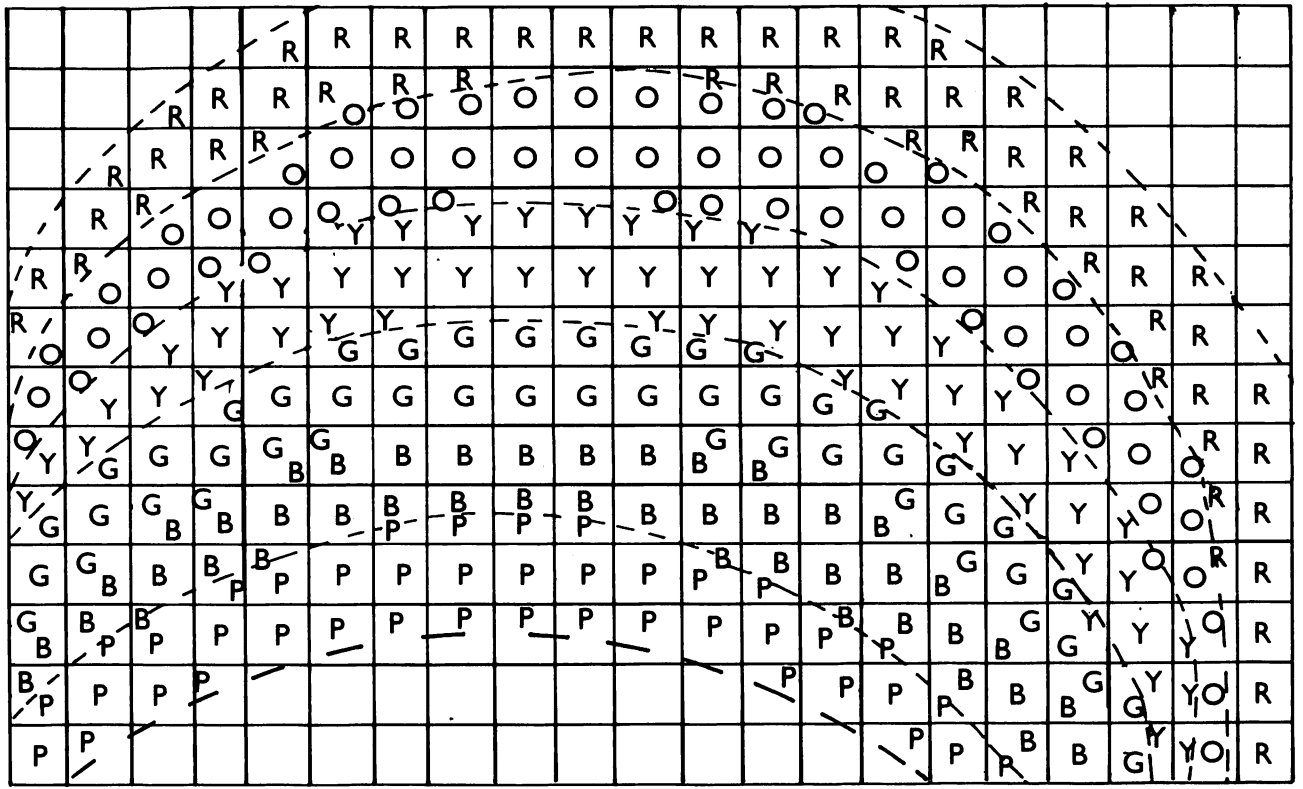
O orange

Y yellow

G green

B blue

P purple



There are certain limitations to the high resolution bit mapped graphics mode when you try to put a lot of colours close together, for instance if a red curve and a blue curve cross on a black background. This can be overcome by trading off a minimal loss in resolution for the ability to put four colours in one character square.

Think of the high resolution graphics screen as follows: Imagine two low resolution (25*40) screens, one in front of the other. Each square, in each screen, can take one of sixteen colours. Take an imaginary pair of scissors and cut out a high resolution shape from the nearest of the two screens. For instance the front screen might be a tree with a brown trunk and green leaves. Only its shape may be high resolution, the divisions between colours in the foreground (or background) must be in low resolution (one colour in each character square).

Now forget about the layers in a high resolution screen and imagine two of them, one in front of the other. The horizontal resolution is halved and two colours are still allowed in each character square. With your imaginary scissors, cut a shape out of the front screen, say the tree. Now the tree itself can be coloured with two colours in each character square, as can the background. This is multicolour mode.

Colour mixing

Unfortunately the colour resolution of the 64 is not as good as it should be, especially on models with early versions of the VIC chips, and when using a television rather than a monitor. This is due, firstly, to there being only three 'grey levels' (other than black and white) associated with the fourteen available colours. These can be seen by displaying all the colours on the screen simultaneously and turning the television colour right down. Colours which have different grey levels can be displayed next to each other satisfactorily, but colours with the same grey level tend to blur into each other. Secondly, if a series of vertical stripes are plotted down the screen, one pixel wide and one pixel apart, an effect known as 'chroma distortion' is produced. Alternate character squares across the screen will have either a red or green tinge. This technique has been used to good effect in the long range scanner in Jeff Minter's 'Attack of the Mutant Camels' program. It is hardly noticeable in multicolour mode, since the pixels are twice as wide as those of high resolution mode.

The first effect can however be used to mix colours together on the screen producing around fifty distinct colours and another fifty or so textures. On most machines colour mixing can be achieved by covering an area on the screen with a narrow (one pixel apart) striped or chequerboard pattern (i.e. setting alternate pixels on and off), so that, for instance, setting the foreground colour to red and the background colour to yellow will produce an orange effect. On the Commodore 64, chroma distortion limits us to

using horizontal stripes, though this does not in itself limit the number of possible mixes, which is restricted by the fact that colour mixing only works if the two colours are taken out of the same grey level set. The sets are as follows (we include black in the dark grey set and white in the light grey set):

Light Grey		Medium Grey		Dark Grey	
No.	Colour	No.	Colour	No.	Colour
1	White	4	Purple	0	Black
3	Cyan	5	Green	2	Red
7	Yellow	8	Orange	6	Blue
13	Light green	10	Pink	9	Brown
15	Light grey	12	Medium grey	11	Dark grey
		14	Light blue		

Thus there are five colours in sets 1 and 3, and six colours in set 2. This gives us a possible total of (5+4+3+2+1)*3+6=51 mixtures (including the normal colour set) and 16*17/2−51=85 textures. These can be used in conjunction with an area fill routine such as the one given in the next two chapters. As an easy exercise, when you have implemented the 'Hires' and 'Multi' programs, try changing the paint routine to allow colour mixing.

Textures

Type in the following program, SAVE it and then RUN it:

```
1 REM"■=SAVE"@0:TEXTURES",8
2 DIMF(19)
4 FORI=0TO19:READF(I):NEXTI
6 DATA0,2,6,9,11,1,3,7,13,15,4,5,8,10,12
,14,0,0,0,0
10 POKE56576,150:POKE53272,8:POKE53265,5
9
20 FORI=24*1024TO32*1024-1STEP2
30 POKEI,255:POKEI+1,0
40 NEXT
50 I=16*1024
60 FORY=0TO24
70 FORX=0TO19
80 IFX>16ORY>16THENC=0
90 IFX<17ANDY<17THENC=F(X)*16+F(Y)
100 POKEI,C:POKEI+1,C:I=I+2
110 NEXTX
```

```
120 NEXTY
140 POKE56576,151:POKE53272,21:POKE53265
,27
```

An array of sixteen by sixteen rectangles is plotted on the screen. Alternate rows of pixels are set to either foreground or background colour. Each row and column of rectangles has one of the sixteen available colours, so that every colour mixture and texture is displayed on the screen simultaneously. The rows and columns are each split into the three sets of grey level colours, so that three large areas where colour mixing works can be seen, one in the top left, one in the centre, and one at the bottom right. The whole array is symmetrical about the leading diagonal (top left to bottom right).

It is interesting to re–order the table of suggested character colour combinations, given on page 152 of the *Reference Manual*, into the three grey level sets as below.

	0	2	6	9	11	1	3	7	13	15	4	5	8	10	12	14
0					*	*	*	*	*	*	*	X	*	*	*	*
2						*		*		X	X		*	*		
6	X					*	*		X	*						*
9						*		*		*			*	*		
11	*					*	X	*	*	*					*	X
1	*	*	*	*	*					*	*	*	X	X	*	*
3	*		*		X							X				X
7	*	*	X	*	*								X	X	*	
13	*		X		*							*				
15	*	*	*	X	*	*					X	X		X	*	X
4	*					X				X						
5	*				X	X	X		*	X						
8	X	*		*		*		*		X						
10	X	*		*		X		X		X						
12	*	X	X	X	*	*				*						
14	*		*		X	*	*			X						

* Excellent
X Fair
' ' Poor

Note that, whereas in the above program all the good mixes occur in the three blocks on the leading diagonal, in this table there are hardly any entries in these three blocks.

5 High Resolution Bit Mapped Mode

In this chapter we implement a set of routines for plotting and drawing pictures on the high resolution bit mapped screen. These can be included in your own programs or used with the driver routine supplied. The whole package forms a keyboard-driven screen editor which can be used for designing detailed layered colour pictures for your own programs. It could easily be extended into a full draughting package.

We jump straight into the implementation of the program, as all the necessary theory has been covered in the last chapter. A detailed explanation of the data compression technique used to store the screen display, and the workings of the area fill routine along with full instructions for use of the package are given at the end of this chapter. In the next chapter we also discuss techniques for drawing lines and circles. By the time you get that far you should be well equipped to implement these in machine code if you so wish.

High resolution screen editor

'Hires' allows the user to produce high resolution colour pictures on a Commodore 64 screen using a set of simple commands. These include draw, erase, invert, colour change, fill (paint) and fast cassette (disc) SAVE and LOAD. A sprite can be moved around the screen, under cursor control, between the foreground and background layers, as a check on the modes of the various objects on the screen. Autorepeat is set on all keys, as usual. By plotting alternate rows of foreground and background colours across the screen colour mixing can be achieved. Remember that if a bit in the bit mapped screen area is set to one, then the corresponding pixel on the screen is set to foreground colour, as defined in colour RAM. If the bit is unset (zero) then the corresponding pixel takes the background colour for that character square.

Hires

Type in the following program. Note that it is in four sections. It is only necessary to type in the first two sections to test simple plotting. The third section contains the cassette handling routines and the fourth section

contains the fill and sprite handling routines. Each section is followed by a line by line description of what is happening and the subroutines can be extracted from the program for use as a graphics package – but remember that the variables, which are initialised in line 20, must always contain their correct values.

'Hires' Program Part One: Listing

```
1 REM"■=SAVE"@0:HIRES",8
5 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■*** HIRES09 **
**"
6 PRINT"■■■■■■■■■■■■■■■■COPYRIGHT (C) 1983
 BY"
7 PRINT"■■■■■■■■KEITH BOWDEN AND PAN BOO
KS LTD"
8 FORI=1TO1000:NEXT
10 POKE 56576,150:POKE 53272,8:POKE 5326
5,59:POKE650,255
20 X=100:Y=100:C=3:S=1:F=1:SX=128:SY=128
:BB=1:A=0:B=1:XS=128
100 BB=-BB:IFBB=1THENGOSUB950
105 GETA$:IFA$=""GOTO100
110 IFA$="I"THENY=Y-S:GOSUB1000:GOTO100:
REM NORTH
111 IFA$="U"THENY=Y-S:X=X-S:GOSUB1000:GO
TO100:REM NORTH WEST
120 IFA$="M"THENY=Y+S:GOSUB1000:GOTO100:
REM SOUTH
121 IFA$="O"THENY=Y-S:X=X+S:GOSUB1000:GO
TO100:REM NORTH EAST
130 IFA$="J"THENX=X-S:GOSUB1000:GOTO100:
REM WEST
131 IFA$=","THENX=X+S:Y=Y+S:GOSUB1000:GO
TO100:REM SOUTH EAST
140 IFA$="K"THENX=X+S:GOSUB1000:GOTO100:
REM EAST
141 IFA$="N"THENX=X-S:Y=Y+S:GOSUB1000:GO
TO100:REM SOUTH WEST
150 IFA$="X"THENGOSUB1100:GOTO100:REM CL
EAR COLOUR
160 IFA$="Z"THENGOSUB1200:GOTO100:REM CL
EAR SCREEN
```

```
170 IFA$="E"THENGOSUB1300:GOTO100:REM EN
D
180 IFA$="C"THENGOSUB1400:GOTO100:REM SE
T COLOURS
190 IFA$="S"THENGOSUB1300:INPUT"STEP";S:
GOSUB1500:GOTO100:REM SET STEP
200 IFA$="F"THENGOSUB1300:INPUT"MODE";F:
GOSUB1500:GOTO100:REM SET MODE
210 IFA$="Q"THENGOSUB2000:GOTO100:REM SA
VE COLOUR
220 IFA$="T"THENGOSUB2100:GOTO100:REM LO
AD COLOUR
230 IFA$="W"THENGOSUB2200:GOTO100:REM SA
VE SCREEN
240 IFA$="R"THENGOSUB2300:GOTO100:REM LO
AD SCREEN
250 IFA$="B"THENGOSUB1500:GOTO100:REM BA
CK TO GRAPHICS
260 IFA$="P"THENGOSUB3000:GOTO100:REM PA
INT (FILL)
270 IFA$=" "THENGOSUB1000:GOTO100:REM NO
THING
280 IFA$="V"THENGOSUB8000:GOTO100:REM TR
IANGLE FILL
300 IFA$="/"THENGOSUB5000:GOTO100:REM SP
RITE
310 IFA$=CHR$( 29)THENSX=SX+8:GOSUB6000:
GOTO100:REM SPRITE EAST
320 IFA$=CHR$(157)THENSX=SX-8:GOSUB6000:
GOTO100:REM SPRITE WEST
330 IFA$=CHR$(145)THENSY=SY-8:GOSUB6000:
GOTO100:REM SPRITE NORTH
340 IFA$=CHR$( 17)THENSY=SY+8:GOSUB6000:
GOTO100:REM SPRITE SOUTH
900 GOTO100
950 G=F:IFF=2THENF=0
990 F=-F:GOSUB1000:F=-F:GOSUB1000:F=G:RE
TURN
```

'Hires' Program Part One: Description

Line 10 Set high resolution bit mapped graphics mode. It is assumed that all registers are set to their defaults at the beginning of the program.
POKE 56576, 150 sets bit 1 to 0 and selects bank 2
POKE 53272, 8 repositions screen
POKE 53265, 59 sets bit 5 to 1 and sets bit mapped mode
POKE 650, 255 sets autorepeat on all keys

20 Initialisation:
X=100: X co-ordinate of paintbrush cursor
Y=100: Y co-ordinate of paintbrush cursor
C=3: Colour code value. C may be:
3 to set foreground and background colours (default)
2 to set background colour only
1 to set foreground colour only
0 to set no colours
S=1: Step between pixels plotted
F=1: Foreground/background code value. F may be:
−1 to erase pixel (set to background colour)
0 to invert pixel
1 to plot pixel (set to foreground colour)
2 for null action
SX=128: set sprite X co-ordinate
SY=128: set sprite Y co-ordinate
BB=1: blink counter for cursor
A=0: set foreground colour
B=1: set background colour
XS=128: temporary sprite co-ordinate

100 Invert blink counter. If 1 then blink cursor

105 Get a character. If none then update blink

110/141 Update cursor co-ordinates. Go to plot subroutine

Lines 150–300 check input values of A$ and perform relevant actions.

Line 150 A$="X": clear colour screen

160 A$="Z": clear bit mapped screen

170 A$="E": set text mode

180 A$="C": set colours

190 A$="S": set text; input step S; set graphics

200 A$="F": set text; input mode F; set graphics

210 A$="Q": save colour screen to cassette

220 A$="T": restore colour screen from cassette

230 A$="W": save bit mapped screen to cassette

240 A$="R": restore bit mapped screen from cassette

250 A$="B": set graphics mode

260	A$="P": call paint (fill) subroutine (see Chapter 7)
280	A$="V": triangle fill
300	A$="/": sprite appear/change colour
310/340	Move sprite in 8 pixel increments
900	Go and get another character, since this one invalid
950	If mode null then set to invert for cursor blink
990	Invert mode; blink cursor; invert mode; restore null

'Hires' Program Part Two: Listing

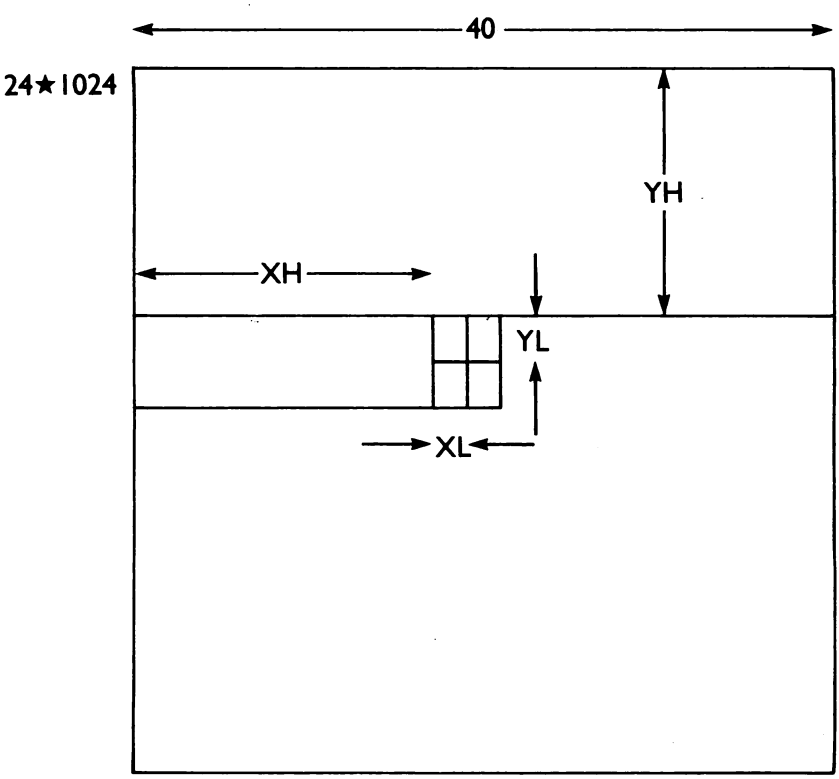
```
1000 IFX<0THENX=0:RETURN:REM PLOT A POIN
T
1001 IFX>319THENX=319:RETURN
1002 IFY<0THENY=0:RETURN
1003 IFY>199THENY=199:RETURN
1005 XI=INT(X):YI=INT(Y)
1010 XH=8*INT(XI/8):YH=8*INT(YI/8)
1015 IFF=2GOTO1055
1020 XL=7+XH-XI
1030 YL=YI-YH
1040 P=24*1024+YL+XH+YH*40
1050 IFF=1THENPOKEP,(PEEK(P))OR(2↑XL)
1051 IFF=-1THENPOKEP,(PEEK(P))ANDNOT(2↑X
L)
1052 IFF=0THENPOKEP,(PEEK(P)ANDNOT(2↑XL)
)OR(NOTPEEK(P)AND(2↑XL))
1055 IFC=0THENGOTO1070
1060 GOSUB1600
1070 RETURN
1100 FOR I=16*1024TO17*1024-1:REM CLEAR
COLOUR
1110 POKE I,1:NEXTI:RETURN
1200 FOR I=24*1024TO32*1024-1:REM CLEAR
SCREEN
1210 POKE I,0:NEXTI:RETURN
1300 POKE56576,151:POKE53272,21:POKE5326
5,27:RETURN:REM SWITCH TO TEXT
1400 GOSUB1300:INPUT"FOREGROUND";A:INPUT
"BACKGROUND";B:REM GET COLOURS
1450 INPUT"CODE";C:GOSUB1500:RETURN
1500 POKE 56576,150:POKE 53272,8:POKE532
65,59:RETURN:REM SWITCH TO GRAPHICS
1600 BYTE=16*1024+YH*5+XH/8:REM SET COLO
US
```

```
1610 IFC=1THENPOKEBY,(PEEK(BY)AND240)ORB
:RETURN
1620 IFC=2THENPOKEBY,(A*16)OR(PEEK(BY)AN
D15):RETURN
1630 IFC=3THENPOKEBY,A*16+B:RETURN
```

'Hires' Program Part Two: Description

Lines 1000–1070 plot a point.

Line 1000/1003 If out of range then return
1005 Store integer parts of cursor co–ordinates
1010 Store high bytes
1015 If mode null then change colour only
1020 Get low byte of X and reverse it
1030 Get low byte of Y
1040 Byte containing pixel at X, Y=start of bit mapped+high byte of Y*40 (*8)+high byte of X (*8)+low byte of Y



Line 1050 If mode plot then set bit XL
1051 If mode erase then unset bit XL
1055 If no colour changes then RETURN
1060/1070 Go to colour change routine and RETURN

1100/1110 Set colour screen to black and white

1200/1210 Set bit mapped screen to background colour

1300 Set text screen: POKE 56576,151 selects bank 1
POKE 53272,21 repositions screen defaults
POKE 53265,27 sets text mode

1400/1450 Set text; get colours and code; set graphics

1500 Switch to graphics mode
1600 Colour byte=start of colour area+high byte of Y * 40+high byte of X
1610 Set background colour only
1620 Set foreground colour only
1630 Set both foreground and background colours

'Hires' Program Part Three: Listing

```
2000 OPEN 1,1,1,"COLOR":REM SAVE COLOUR
2010 FOR I=1024*16TO1024*17-1
2020 PRINT#1,PEEK(I)
2030 NEXT I:CLOSE1:RETURN
2100 OPEN 1,1,0,"COLOR":REM LOAD COLOUR
2110 FOR I=1024*16TO1024*17-1
2120 INPUT#1,Q:POKE I,Q
2130 NEXT I:CLOSE1:RETURN
2200 OPEN 1,1,1,"PICTURE":REM SAVE SCREE
N
2210 CO=0
2220 FORI=1024*24TO1024*32-1
2230 A%=PEEK(I)
2240 IFCO=0ANDNOT(A%=0ORA%=255)THENPRINT
#1,CHR$(A%);:GOTO2295
2250 IFCO=0THENB%=A%:CO=1:GOTO2295
2260 IFA%=B%THENCO=CO+1:GOTO2295
2270 IFB%=0THENCO=-CO
2271 PRINT#1,CHR$(255);CO:CO=0
2280 IFA%=0ORA%=255THENB%=A%:CO=1:GOTO22
95
2290 PRINT#1,CHR$(A%);
2295 NEXTI:IFCO=0GOTO2299
```

```
2296 IFB%=0THENCO=-CO
2297 PRINT#1,CHR$(255);CO
2299 CLOSE1:RETURN
2300 OPEN1,1,0,"PICTURE":REM LOAD SCREEN
2310 FORI=1024*24TO1024*32-1
2320 GET#1,A$:IFA$=""GOTO2320
2325 A%=ASC(A$)
2330 IFA%<>255THENPOKEI,A%:GOTO2350
2335 INPUT#1,CO:IFCO<0THENCO=-CO:A%=0
2340 FORJ=1TOCO:POKEI+J-1,A%:NEXTJ:I=I+C
O-1
2350 NEXTI:CLOSE1:RETURN
```

'Hires' Program Part Three: Description

Lines 2000–2350 control cassette handling. To convert to disc change line 2000 to read:

OPEN 1, 8, 8, "COLOR, S, W": REM SAVE COLOR

Line 2000/2030 Save colour screen to cassette: Open file called COLOR on cassette, transfer contents of colour screen to cassette as ASCII decimal code

2100/2130 Load in colour screen from cassette

Lines 2200–2299 save the bit mapped screen to cassette:
A% is value of current byte
B% is value of last byte
CO is count of bytes set to 0 or 255 (positive for 255 bytes, negative for zero bytes)

Line 2200/2210 Open cassette file PICTURE; set CO
2220 Bit mapped screen area
2230 Get byte
2240 If not counting special bytes (set to 0 or 255) and current byte not 0 or 255 then print byte to file and restart loop
2250 If not counting special bytes then save current byte, set count to 1 and restart loop
2260 If special byte same as last time then increment count and restart loop
2270 If zero byte then negate count
2271 Print byte followed by count ; reset count

2280	If special byte then save it. Set count to 1 and restart loop
2290	Print byte
2295	Restart loop. At end of loop test for zero count and jump to end
2296	If zero byte then negate count
2297	Print byte followed by count
2299	Close file and RETURN

Lines 2300–2350 load bit mapped screen file from cassette

Line 2300	Open cassette file PICTURE
2310	Loop through bit mapped screen
2320	Get a character; if null try again
2325	Convert to decimal
2330	If not 255 restore it to screen and start again
2335	Otherwise read count; if negative then negate it and set byte to zero
2340	Restore CO bytes to screen; increment I by (count–1)
2350	End loop. Close file. RETURN

'Hires' Program Part Four: Listing

```
3000 XD=1:REM FILL ROUTINE
3010 XREF=X:YREF=Y
3020 GOSUB4000
3025 GOSUB1600
3030 IFXD=1THENIFX=XHANDPEEK(P)=0THENPOK
EP,255:X=X+8:GOTO3020
3040 IFXD=-1THENIFX=XH+7ANDPEEK(P)=0THEN
POKEP,255:X=X-8:GOTO3020
3055 IFLO=0THENPOKEP,(PEEK(P))OR(2↑XL):X
=X+XD:GOTO3020
3060 XD=-XD:X=XREF:IFXD=-1GOTO3020
3070 X=X+1:Y=Y+1:GOSUB4000
3080 IFLO=0THENPOKEP,(PEEK(P))OR(2↑XL):X
=X+XD:GOTO3020
3090 X=XREF:Y=YREF:RETURN
4000 XH=8*INT(X/8):REM GET A PIXEL
4010 YH=8*INT(Y/8)
4020 XL=7+XH-X
4030 YL=Y-YH
4040 P=24*1024+YL+XH+YH*40
4050 LO=(PEEK(P))AND(2↑XL)
4060 RETURN
```

```
5000 SS=SS+1:IFSS/2=INT(SS/2)THEN5050:RE
M SET SPRITE
5010 POKE53271,1:POKE53277,1
5020 POKE53248,XS:POKE53249,SY:POKE53287
,SC:POKE53269,1:POKE53275,1
5025 POKE16384+2040-1024,32:FORI=16384+3
2*64TO16384+33*64:POKEI,255:NEXT
5030 SC=SC+1:IFSC=16THENSC=0
5040 RETURN
5050 POKE53269,0:RETURN
6000 IFSX<0THENSX=0:RETURN:REM MOVE SPRI
TE
6001 IFSX>319THENSX=319:RETURN
6002 IFSY<0THENSY=0:RETURN
6003 IFSY>255THENSY=255:RETURN
6004 IFSX>255THENPOKE53264,1:XS=SX-256
6005 IFSX<256THENPOKE53264,0:XS=SX
6010 POKE53248,XS:POKE53249,SY:RETURN
8000 V=V+1:XT(V)=X:YT(V)=Y:REM TRIANGLE
FILL
8010 IFV<>3THENRETURN
8020 Y1=YT(1):Y2=YT(2):Y3=YT(3):X1=XT(1)
:X2=XT(2):X3=XT(3):V=0
8030 IFY2<Y1THENY2=Y1:X2=X1:Y1=YT(2):X1=
XT(2)
8040 IFY3<Y2THENY3=Y2:Y2=YT(3):X3=X2:X2=
XT(3)
8050 IFY2<Y1THENYT(2)=Y2:XT(2)=X2:Y2=Y1:
X2=X1:Y1=YT(2):X1=XT(2)
8120 M3=(X3-X1)/(Y3-Y1):C3=Y3*M3-X3:REM
Y3#Y1
8125 SP=1:IFX2>Y2*M3-C3THENSP=-1
8128 IFY2=Y1THEN8158
8129 M1=(X2-X1)/(Y2-Y1):C1=Y1*M1-X1
8130 FORY=Y1TOY2
8140 FORX=Y*M1-C1TOY*M3-C3STEPSP
8150 GOSUB1000:NEXTX,Y
8158 IFY3=Y2THEN8190
8159 M2=(X3-X2)/(Y3-Y2):C2=Y2*M2-X2
8160 FORY=Y2TOY3
8170 FORX=Y*M2-C2TOY*M3-C3STEPSP
8180 GOSUB1000:NEXTX,Y
8190 RETURN
```

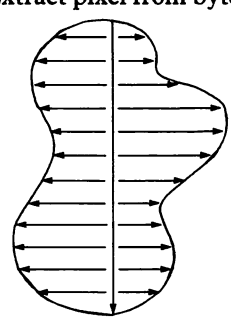
'Hires' Program Part Four: Description

Lines 3000–3090 are the fill (paint) subroutine

Line		
	3000	Set X increment XD equal to 1 to scan right
	3010	Save initial co-ordinates
	3020	Get a pixel value in LO
	3025	Set colours
	3030	If scanning to the right and next byte is zero then set to 255; increment 8 pixels; go on to next byte (start of byte only)
	3040	If scanning left and next byte zero then set to 255; decrement 8 pixels; go on to next byte (start of byte only)
	3055	If current pixel not set then set it (XLth pixel in current byte); increment X; go on to next pixel.
	3060	Otherwise we have reached boundary; reverse scan direction; restore X co-ordinate; if scanning to the left go on to next pixel
	3070	If scanning to the right increment Y (go on to next row) and X (to avoid duplication); get another pixel
	3080	If pixel not set, set it; increment X; go on to next pixel
	3090	Otherwise we must have scanned down to lower boundary: RETURN to calling program

Lines 4000–4060 get a pixel (routine called by fill)

Line		
	4000/4010	Get high bytes
	4020/4030	Get low bytes (reverse X)
	4040	Get current byte address
	4050/4060	Extract pixel from byte; RETURN



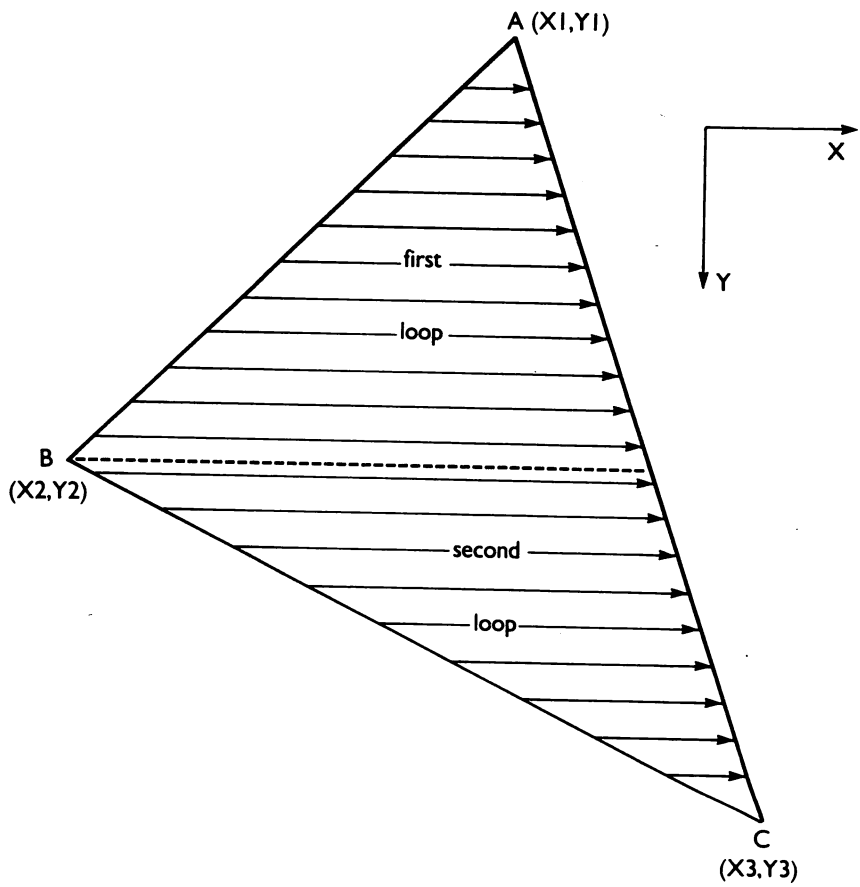
Lines 5000–5050 set up and turn off sprite

Lines 6000–6010 deal with sprite handling

Line		
	6000/6001	Set limits to sprite position
	6004/6005	If X co-ordinate greater than 255 then set MSB register to 1
	6010	Set sprite X and Y co-ordinates

Lines 8000–8190 Triangle fill (every third "V" keyed in)

Line	8000	Increment corner; save X and Y
	8010	If not third corner then return
	8020	Transfer arrays to variables; reset counter
	8030/8050	Order points in Y dimension so that Y1<=Y2<=Y3
	8120	M3 is gradient of line between points 1 and 3; C3 is point at which line crosses X axis Equation of line is X3=Y3*M3+C3, and Y3 can never be equal to Y1 for a triangle with finite area (think about it), so the equation can never overflow
	8125	Set step to 1. If X2 on right hand side of AC then scan from right to left



	8128	If Y2=Y1 then omit first loop
	8129	Calculate equation of AB
	8130	Top half of triangle
	8140	From AB to AC scan left to right

8150	Set pixel; repeat; repeat This routine could be made more efficient if every internal zero byte was set to 255 using one POKE, in the same way as done in the fill routine
8158	If Y2=Y3 then omit this loop
8159	Calculate equation of BC
8160	Bottom half of triangle
8170	From BC to AC scan left to right
8180	Set pixel (see comments above); repeat
8190	RETURN to calling program

'Hires' Program: Controls

Paintbrush controls:

I	Move paintbrush S pixels to the North (up)
U	Move paintbrush S pixels to the North West
M	Move paintbrush S pixels to the South (down)
O	Move paintbrush S pixels to the North East
J	Move paintbrush S pixels to the West (left)
,	Move paintbrush S pixels to the South East
K	Move paintbrush S pixels to the East (right)
N	Move paintbrush S pixels to the South West

Note that for convenience the eight control keys chosen are arranged approximately as the points of a compasS:

U I O

J K

N M ,

Autorepeat is set on all keys so that, with the normal defaults set, holding down the "J" key will draw a horizontal line to the left.

Clear Screen controls:

X	Clear colour to black foreground and white background
Z	Clear bit mapped screen to background colour

Mode controls:

E	Switch to text mode (end)
B	Switch to graphics mode (begin)

Screen Save controls:

Q Save colour screen to cassette tape
T Restore colour screen from cassette tape
W Save bit mapped screen to cassette tape
R Restore bit mapped screen from cassette tape

Sprite controls:

These are designed to allow you to check which parts of the screen are foreground and which background:

/ First depression: sprite appears (black)
Second depression: sprite disappears
Third depression: sprite appears (white)
Fourth depression: sprite disappears
Fifth depression: sprite appears (red)
. . . and so on.

Every odd depression makes a large rectangular sprite appear, cycling through the palette of sixteen colours. Every even depression makes the sprite disappear.

▲ Move sprite up eight pixels
▼ Move sprite down eight pixels
▶ Move sprite right eight pixels
◀ Move sprite left eight pixels

Fill Routine controls:

V Triangle fill. This command cycles through three key depressions. On the first and second the current co-ordinate of the paintbrush is stored. Every third depression the triangle fill routine uses the current and the two stored co-ordinates to define a triangle, which is then filled according to the current paintbrush attributes. Remember to move the paintbrush between key depressions!

P Paint. Fills an area enclosed by pixels set to foreground colour with the foreground colour. The algorithm is simple. The program starts at the current point and moves downwards, whilst searching left and right for set pixels, filling the area as it goes. When it reaches the first set pixel vertically below the start, it stops and returns to the start point. Thus only simply connected areas can be filled, and it may be necessary to use the routine more than once to fill a complicated shape.

Parameter controls:

The screen is automatically switched into text mode before a parameter is set, and out after setting. When the appropriate key is pressed the program prompts for numeric input, after which the RETURN key must be pressed.

C Set colours. The program prompts for Foreground colour and Background colour, each in the range 0–15. Note these are not the colours given on the numeric keyboard. The colours are specified as:

0	Black	8	Orange
1	White	9	Brown
2	red	10	Pink
3	Cyan	11	Light grey
4	Purple	12	Grey
5	Green	13	Light green
6	Blue	14	Light blue
7	Yellow	15	Dark grey

The program then prompts for colour Code (colour screen only), in the range 0–3:

0	Do not change colours whilst plotting
1	Change only background colour
2	Change only foreground colour
3	Change both colours

S Step. Number of pixels to step while plotting. If set to any positive value other than 1, will produce a dotted line. Normally used to move quickly around screen without plotting or to set up a grid.

F Paintbrush attribute (bit mapped screen only), in the range −1 to 2:

−1	Plot pixel in background colour (erase)
0	Plot pixel in inverse colour (invert)
1	Plot pixel in foreground colour
2	Do not change pixel (for moving paintbrush)

In invert mode it is necessary to press the space bar when reversing direction to avoid leaving an odd pixel (each pixel must be converted twice).

Using 'Hires'

Whether or not you have typed in the last two sections of the program, type RUN and you should see the copyright notice, followed by a set of random blocks, appearing on the screen. On my machine they are black, white and purple. Hit the Z key and they are cleared to black and white, X and the whole screen turns white. A small flashing cursor is seen near the centre of the screen. This is your paintbrush. Using the cursor controls given in the table you can move the cursor around the screen, leaving a trailing line, Etch-a-Sketch fashion. Angles of 45 and 90 degrees are available. To produce a smooth curve, different angles can be drawn by alternating key–presses between two keys, and it is possible to draw a decent circle. Try tracing a line drawing onto tracing paper. Stick this over your television screen and follow the lines around with the cursor. Ignore the colour fringes, as these are the 'chroma distortion' effect described in the last chapter. The only thing you can do about them is to turn the colour intensity on your television down, in which case they disappear, or choose a better colour combination than black and white; for example, light and dark blue, light and dark green, or (my favourite), orange on brown. Remember that thin vertical black lines on a white background are likely to end up with a red or green tinge.

To trace a picture onto the screen you need to be able to move the cursor without drawing a line, and also erase anything you have drawn incorrectly. To do this hit the F key. The screen will switch to text and you will be prompted to input a number. The codes are given in the parameters control table. Try these whilst you are drawing your picture. Note that the screen switches back to graphics as soon as you hit the RETURN key after entering a number and that the cursor still flashes, in any mode and at any position on the screen. The only time you will lose it is if it is within a character square in which the foreground and background colours are the same.

To change colour press C. You will be prompted for the foreground colour, background colour and a code to say which of these you actually want to change when plotting. (Often it will be just the foreground.) Experiment with this facility. It might seem complicated, but it is very powerful. Read the comments on use of colour in the last chapter very carefully, until you can relate what is happening on the screen to what the program is doing with your input. You should soon be able to design quite sophisticated pictures. To start again press X and Z (or just one of them). To have a look at the text screen press E, and use B to return to graphics. If at any time the program seems to have crashed (it is most unlikely that it actually has) press STOP/RESTORE and type RUN. The cursor will be reset to the middle of the screen and all the parameters will be reset to their defaults, but your picture will still be there. To move the cursor rapidly around the screen, or to set background colours only, set S to 8 or any other suitable value.

For what follows it is necessary to have the whole program keyed in and running. First try triangle fill. Position the cursor and press the V key. Position the cursor again and press V again. Reposition the cursor such that the current position and the last two points are not in a straight line, and press V a third time. The three points form a triangle, which is filled in according to the settings of the mode (F), colour code (C), and foreground and background colours. For instance, if the triangle is drawn in invert mode, and it initially contained a small object in foreground colour, then when the full routine has finished the object will be in background colour, with a surround of foreground colour! Thus this is a very general routine, though rather slow. It would be possible to convert it such that if it was necessary to change all bits in a byte from 0 to 1, the byte in question would simply be POKEd with 255. This could speed things up tremendously, as can be seen with the paint routine described below. The triangle routine contains, inherently, a line drawing routine, which could easily be implemented separately. To plot a line it is only necessary to know its equation, $y=mx+c$, where m is the 'gradient' of the line and c the point at which it crosses the y axis. These constants can be found by substituting the co-ordinates of the ends of the line into the equation, thus producing two equations in two unknowns.

Now for a routine that paints in general outlines. Draw a shape in foreground colour and position the cursor one pixel below the top of the shape. Press P. Some or all of the shape will be filled in. The routine scans left and right, looking for a set pixel, and filling at the same time. When it has completed one row it moves down a pixel and does the next one (see the diagram in the program explanation). When it reaches a set pixel vertically below the starting point it stops, and returns the cursor to the start point. Thus the routine will only fill simply connected shapes, and even then it may be necessary to reposition the cursor and press P again, if the routine leaves behind an unfilled area. Sophisticated fill routines 'remember' the bits they have left behind and go back and fill them in. Nevertheless this routine is quite efficient (for BASIC), as whenever it can it fills blocks of eight horizontal pixels. You should be able to see it accelerate when it moves away from a boundary towards the centre of an object. This paint routine is less general than the triangle fill, though, as it will only fill an area defined by a foreground colour outline, with foreground colour. Many variations on this are possible, as we will see in the next chapter. Another type of fill routine starts from a point somewhere in the middle of the area, and goes round in an ever-growing diamond shaped spiral until it has covered the whole area (or screen). Every time it comes across a set pixel it switches the fill between foreground and background plotting.

You should now have a nice mess on the screen. Press the / key. A double size black sprite appears in the middle of the screen. (Big, isn't it?) Press / again and it disappears. Again, and it reappears a different colour. Keep on

pressing until it becomes a colour you have not used before. Using the cursor control keys you can move the sprite around, even on to the right hand side of the screen. Notice something? Yes, the sprite appears to move over the screen background colours, but *under* the foreground colours. Using this facility you can check which parts of the screen are drawn in which mode, for use with sprites in a later program. See how the foreground is high resolution but the background is low resolution, and refer to the previous chapter if you don't follow the process.

Finally, try saving and restoring the screen. Remember that the bit mapped screen and the colour area are stored separately. The screen restore routines can be included in your own programs when you have designed your picture and stored it.

Saving the screen

To save a bit mapped screen on to disc or tape it is necessary to store around 9K of memory, including the colour RAM. Simply performing a hexadecimal dump takes around a minute on disc or seven minutes on tape, with a similar amount of time taken to restore the screen. My original program wrote decimal numbers to tape and took over fifteen minutes to save a picture! As these times are inconveniently long it is worth looking for a way of compressing the data.

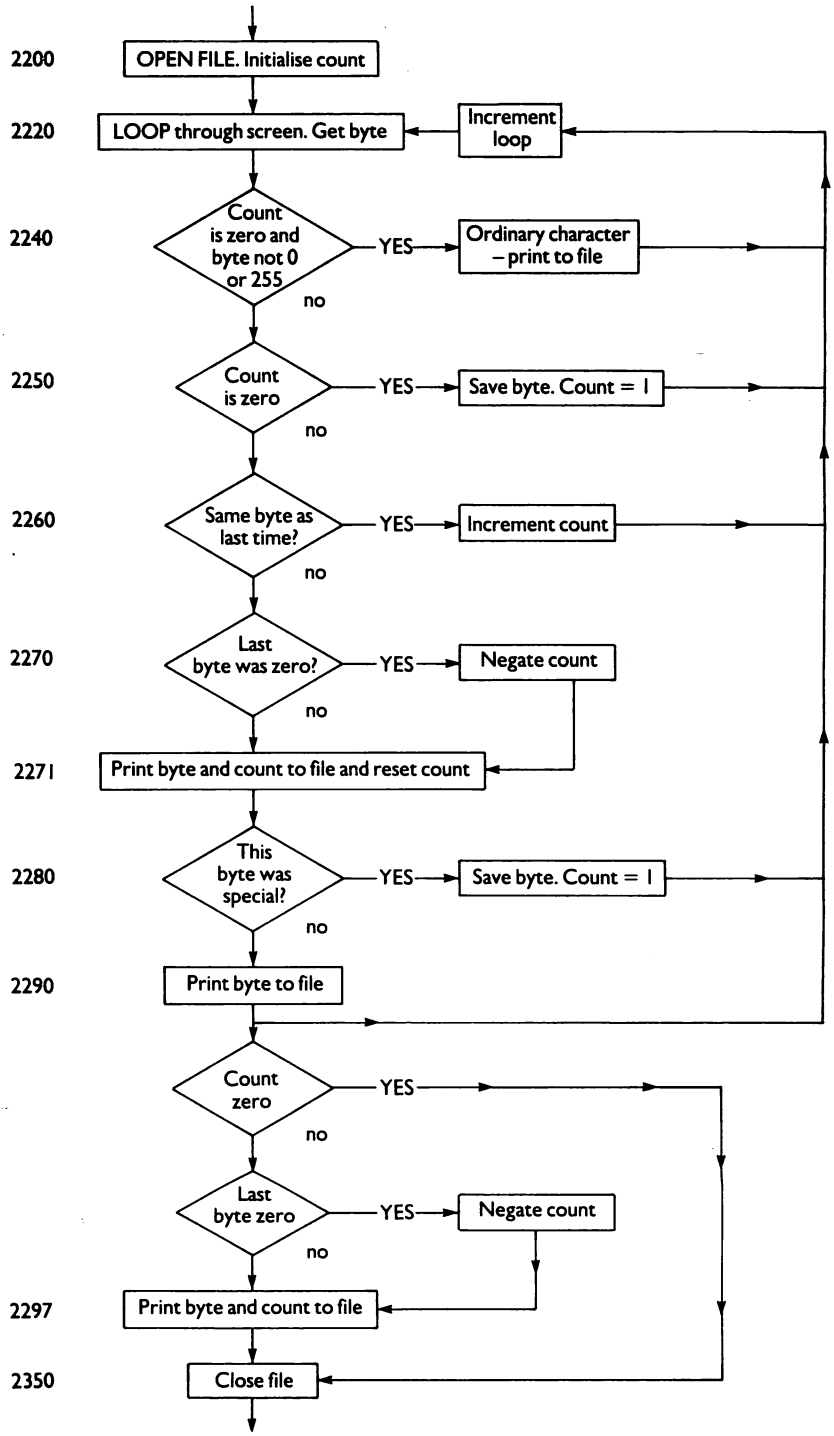
The algorithm used in the programs 'Hires' and 'Multi' assumes that the picture you are saving is relatively simple, and hence contains long sequences of bytes set to either 0 or 255. Any values other than these are saved as their ASCII equivalents, taking up the same amount of storage as they would from a hex dump. When a value of 0 or 255 is encountered the algorithm counts the number of consecutive zeroes (or 255s) in the screen area and saves a byte of value 255. This count is set positive for 255s and negative for 0s. Note that zero (null) bytes cannot be saved to serial files as they are used as the end of record terminator. (Actually they can be saved but not read back from BASIC.) A flow chart of this data compression method is shown below. Also study the technique used to restore the screen. Note that the colour screen still uses my original inefficient technique. This matters little as colour RAM is only 12% of the size of the bit mapped screen. However as an exercise you could find a more efficient algorithm for saving and restoring it. The technique used for the bit mapped screen would not be particularly suitable. At a pinch a hex dump can be achieved from BASIC by setting the start and end of BASIC pointers (see the memory map in Appendix A) to the start and end of the screen area and performing an ordinary hex dump. Note that null characters *can* be saved this way.

To convert the programs to disc, every occurrence of OPEN 1, 1, 1 should be changed to OPEN 1, 8, 8 and the filename (e.g. "COLOR") should have ",S,W"

added if a write file is required, or ", S, R" for a read file. See p.19 of *Disk User's Manual* for more details.

An alternative method that could be used (machine code is a necessity) involves a two dimensional binary chop of the bit mapped screen. In this way we could perform a disc save in a few seconds! The technique essentially involves dividing the screen into four quarters and checking whether each quarter contains any set pixels. If it does not, it is ignored for now; if it does then the quarter itself is divided into quarters and the process repeated until one quarter is equal to one pixel. (Obviously this is easier with a 256×256 screen.) The structure you end up with can be represented as a 'tree' and saved and restored very efficiently. Recent work on this subject shows that interesting transformations such as rotation can be very efficiently performed on the tree structure itself. An article on data compression techniques is given in Reference 2. Remember – you are saving not only time but also disc space.

Compressing the Bit Mapped Screen



Calling the 'Hires' routines from your own programs

PLOT: Plots a point. Lines 1000–1070, 1600–1630
Calls SET COLOUR routine.
Usage: Set X to X co-ordinate of point (0<=X<=319)
Y to Y co-ordinate of point (0<=Y<=199)
F to −1, 0, 1, 2 for erase, invert, plot or null
C to 0, 1, 2, 3 for colour change (bit 1 change background colour, bit 2 change foreground colour)
A to foreground colour (0–15)
B to background colour (0–15)
Call: GOSUB1000
Returns: XH, XL, YH, YL, giving low and high bytes of X and Y co-ordinates
P and BY give location of pixel and colour bytes

CLEAR COLOUR: Clears colour RAM. Lines 1100–1110
Call: GOSUB1100

CLEAR SCREEN: Clears bit mapped screen. Lines 1200–1210
Call: GOSUB1200

SET TEXT: Switches to text mode. Line 1300
Call: GOSUB1300

SET GRAPHICS: Switches to graphics mode. Line 1500
Call: GOSUB1500

SAVE COLOUR: Saves colour RAM. Lines 2000–2030
Call: GOSUB2000

SAVE PICTURE: Saves bit mapped screen. Lines 2200–2299
Call: GOSUB2200

LOAD COLOUR: Restores colour RAM. Lines 2100–2130
Call: GOSUB2100

LOAD PICTURE: Restores bit mapped screen. Lines 2300–2350
Call: GOSUB2300

FILL: Paints an enclosed area in foreground colour. Lines 3000–4060
Calls FETCH PIXEL and COLOR routines
Usage: Set X and Y to top centre of enclosed area
Set A, B, C as for PLOT routine
Call: GOSUB3000
Returns: As PLOT and FETCH PIXEL

FETCH PIXEL: Gives value of specified pixel. Lines 4000–4060
Usage: Set X and Y to current co-ordinates
Call: GOSUB4000
Returns: Value of pixel in LO

TRIANGLE FILL: Fills a triangle according to A, B, C, F, X and Y values. (Must be called three times with different values of X and Y to define corners of triangle.)
Usage: Set X and Y to current co-ordinates
Set F and C as for PLOT routine
Set A and B as for PLOT routine

6 Multicolour Bit Mapped Mode

In this chapter we implement a set of routines for plotting and drawing pictures on the multicolour bit mapped screen, and a keyboard driver for the routines similar to that of the last chapter. At the end of the chapter we discuss techniques for plotting lines and circles.

A multicolour screen editor

Remember that multicolour mode only has a resolution of 160×200 pixels, but allows two foreground colours and two background colours in each character square. One foreground colour and one background colour are obtained from the colour RAM in the VIC area. One background colour is constant over the whole screen (screen colour) and the other foreground colour is obtained from the colour nybble area. It is necessary to be very careful with this second foreground colour, as it will scroll up with the BASIC text screen. In the program which follows we prevent this by never allowing the screen to scroll. It still, however, changes with the text screen.

We also introduce a slightly different system of controlling plotting. The mode of the paintbrush can still be set to plot, erase, invert or null, but these only control plotting between alternative colours *on a particular level*. An extra mode called 'level' can also be set to plot, erase, invert or null (still 1, 0, −1 or 2), and controls plotting between levels. If the level is set to 0, foreground will thus be converted to background, and vice versa. If mode is set to 0 and level to 2 then a pixel on a particular layer will have its colour changed *to the other colour for that layer*.

The usual sprite is available for differentiating between layers. This emphasises that both background and foreground plotting are now available in high resolution. The 'paint' routine differs from that of the last chapter. An area is now filled up to a boundary *with exactly the same mode as the pixel currently being plotted*. Thus if the paintbrush is set to layer 2, colour 1, then filling will occur with this mode, and up to a boundary consisting of set pixels using this mode. Any pixels set to other modes within the area will simply be overwritten, in the same way as objects within a triangle fill area are treated by the program in the last chapter.

It is intended that the user *designs his picture from the back, forwards*. So, first set screen colour, draw any objects in background colour 2 and fill them

in, next draw objects in foreground colour 1, fill them in and lastly draw objects and fill them in foreground colour 2. If any other technique is used it is probably better to use a fill routine equivalent to the one in 'Hires', which just fills up to a boundary of arbitrary mode. This is left as an exercise for the reader. The triangle fill routine generalises as one might expect. A more robust (but difficult to follow) version is given.

'Multi'

Type in the program section by section (again there are four of them), while following the line by line explanation after each section listing. If you like, you may implement the program by editing the 'Hires' program in the previous chapter. Should you choose this approach exercise great care, as there are many subtle differences. This is the reason we present the program again in its entirety. As always you can include the routines in your own program, provided you ensure that all the variables initialised in line 20 always contain their correct values. The explanations of most lines are not as detailed as those in the last chapter, as there is no point in repeating ourselves. If any problems arise, refer back.

'Multi' Program Part One: Listing

```
1 REM"■=SAVE"@0:MULTI",8
5 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒*** MULTI25 **
*"
6 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒COPYRIGHT (C) 1983
 BY"
7 PRINT"▒▒▒▒▒▒▒▒KEITH BOWDEN AND PAN BOO
KS LTD"
8 FORI=1TO1000:NEXT
10 POKE 56576,150:POKE 53272,8:POKE 5326
5,59:POKE53270,216:POKE650,255
15 E=PEEK(53281):PRINT"▒":GOSUB1105
20 X=100:Y=100:C=0:S=1:F=1:L=1:D=14:BB=1
:SX=128:SY=128:A=1:B=0:E=6:XS=128
100 BB=-BB:IFBB=1THENGOSUB950
105 GETA$:IFA$=""GOTO100
110 IFA$="I"THENY=Y-S:GOSUB1000:GOTO100:
REM NORTH
111 IFA$="U"THENY=Y-S:X=X-S*2:GOSUB1000:
GOTO100:REM NORTH WEST
120 IFA$="M"THENY=Y+S:GOSUB1000:GOTO100:
REM SOUTH
121 IFA$="O"THENY=Y-S:X=X+S*2:GOSUB1000:
GOTO100:REM NORTH EAST
```

```
130 IFA$="J"THENX=X-S*2:GOSUB1000:GOTO10
0:REM WEST
131 IFA$=","THENX=X+S*2:Y=Y+S:GOSUB1000:
GOTO100:REM SOUTH EAST
140 IFA$="K"THENX=X+S*2:GOSUB1000:GOTO10
0:REM EAST
141 IFA$="N"THENX=X-S*2:Y=Y+S:GOSUB1000:
GOTO100:REM SOUTH WEST
150 IFA$="X"THENGOSUB1100:GOTO100:REM CL
EAR COLOUR
160 IFA$="Z"THENGOSUB1200:GOTO100:REM CL
EAR SCREEN
170 IFA$="E"THENGOSUB1300:GOTO100:REM EN
D
180 IFA$="C"THENGOSUB1400:GOTO100:REM SE
T COLOUR
190 IFA$="S"THENGOSUB1300:PRINT"[illegible]
[illegible]STEP ";S;:INPUTS:GOSUB1500:GOTO100
200 IFA$="F"THENGOSUB1300:PRINT"[illegible]
[illegible]MODE ";F;:INPUTF:GOSUB1500:GOTO100
210 IFA$="Q"THENGOSUB2000:GOTO100:REM SA
VE COLOUR
220 IFA$="T"THENGOSUB2100:GOTO100:REM LO
AD COLOUR
230 IFA$="W"THENGOSUB2200:GOTO100:REM SA
VE SCREEN
240 IFA$="R"THENGOSUB2300:GOTO100:REM LO
AD SCREEN
250 IFA$="B"THENGOSUB1500:GOTO100:REM BE
GIN
260 IFA$="P"THENGOSUB3000:GOTO100:REM PA
INT (FILL)
270 IFA$=" "THENGOSUB1000:GOTO100:REM NO
THING
280 IFA$="V"THENGOSUB8000:GOTO100:REM TR
IANGLE FILL
290 IFA$="L"THENGOSUB1300:PRINT"[illegible]
[illegible]LAYER";L;:INPUTL:GOSUB1500:GOTO100
300 IFA$="/"THENGOSUB5000:GOTO100:REM SP
RITE
310 IFA$=CHR$( 29)THENSX=SX+8:GOSUB6000:
GOTO100:REM SPRITE EAST
320 IFA$=CHR$(157)THENSX=SX-8:GOSUB6000:
GOTO100:REM SPRITE WEST
```

```
330 IFA$=CHR$(145)THENSY=SY-8:GOSUB6000:
GOTO100:REM SPRITE NORTH
340 IFA$=CHR$( 17)THENSY=SY+8:GOSUB6000:
GOTO100:REM SPRITE SOUTH
900 GOTO100
950 G=F:IFF=2THENF=0
990 F=-F:GOSUB1000:F=-F:GOSUB1000:F=G:RE
TURN
```

'Multi' Program Part One: Description

Line 10 Set bank 2; reposition screen; set bit mapped mode; POKE 53270,216 sets multicolour mode (bit 5 set to 1); set autorepeat on all keys

15 E set equal to screen colour (background colour 1); clear screen; clear colour nybble area to light blue

20 Set default values of global variables:

cursor co-ordinates	Y=1000
	X=1000
colour code	C=0
cursor step	S=1
paintbrush colour mode	F=1
paintbrush level mode	L=1

So the default is 'plot in foreground colour 2' (colour nybble area), which is dark blue

foreground colour 2	D=14 (light blue)
sprite co-ordinates	SX=128
	SY=128
background colour 2	B=0 (black)
foreground colour 1	A=1 (white)
background colour 1	E=6 (dark blue)
temporary sprite co-ordinate	XS=128

100/105 Flash cursor; get character

110/141 Update co-ordinates (X is always incremented or decremented by double the step length, and is always even)

150/900 Other controls including layer, L; RETURN
Note that the text cursor is always repositioned for parametric input

950/990 Flash cursor

'Multi' Program Part Two: Listing

```
1000 IFX<0THENX=0:RETURN:REM PLOT A POIN
T
1001 IFX>319THENX=319:RETURN
1002 IFY<0THENY=0:RETURN
1003 IFY>199THENY=199:RETURN
1005 XI=INT(X):YI=INT(Y)
1010 XH=8*INT(XI/8):YH=8*INT(YI/8)
1020 XL=7+XH-XI
1030 YL=YI-YH
1040 P=24*1024+YL+XH+YH*40
1045 IFL=1THENPOKEP,(PEEK(P))OR(2↑(XL))
1046 IFL=-1THENPOKEP,(PEEK(P))ANDNOT(2↑(
XL))
1047 IFL=0THENPOKEP,(PEEK(P)ANDNOT(2↑(XL
)))OR(NOTPEEK(P)AND(2↑(XL)))
1050 IFF=1THENPOKEP,(PEEK(P))OR(2↑(XL-1)
)
1051 IFF=-1THENPOKEP,(PEEK(P))ANDNOT(2↑(
XL-1))
1052 IFF=0THENPOKEP,(PEEK(P)ANDNOT(2↑(XL
-1)))OR(NOTPEEK(P)AND(2↑(XL-1)))
1060 GOSUB1600
1070 RETURN
1100 FOR I=16*1024TO17*1024-1:REM CLEAR
COLOUR
1102 POKE I,1:NEXTI
1105 FOR I=55296TO56295
1110 POKE I,14:NEXTI:RETURN
1200 FOR I=24*1024TO32*1024-1:REM CLEAR
SCREEN
1210 POKE I,0:NEXTI:RETURN
1300 POKE56576,151:POKE53272,21:POKE5326
5,27:POKE53270,200:RETURN:REM TEXT MODE
1400 GOSUB1300:PRINT"■■■■■BACK1 (";E;")
";:INPUTE:POKE53281,E:REM GET COLOURS
1401 PRINT"■■■BACK2 (";B;")";:INPUTB
1405 PRINT"■■■FORE1 (";A;")";:INPUTA
1410 PRINT"■■■FORE2 (";D;")";:INPUTD
1420 PRINT"■■■CODE  (";C;")";:INPUTC
1430 GOSUB1500:RETURN
1500 POKE 56576,150:POKE 53272,8:POKE532
65,59:POKE53270,216:RETURN:REM GRAPHICS
1600 BY=16*1024+YH*5+XH/8:REM SET COLOUR
```

```
1610 IFCAND1=1THENPOKEBY,(PEEK(BY)AND240
)ORA
1620 IFCAND2=2THENPOKEBY,(B*16)OR(PEEK(B
Y)AND15)
1630 IFCAND4=4THENPOKE55296+YH*5+XH/8,D
1640 RETURN
```

'Multi' Program Part Two: Description

Lines 1000–1070 plot a multicolour point

Line 1000/1040 Keep cursor on screen; get high and low bytes of co-ordinates and calculate memory location of current pixel
1045 Set pixel to foreground
1046 Set pixel to background (odd bits)
1047 Invert pixel layer
1050 Set pixel to colour 2
1051 Set pixel to colour 1 (even bits)
1052 Invert pixel colour
1060/1070 Plot colour; RETURN

1100/1102 Clear colour RAM to black and white
1105/1110 Clear colour nybbles to light blue
1200/1210 Clear bit mapped screen to background colour 1
1300 Set text mode; POKE 53270, 200; Set hires text mode
1400/1430 Get colours and code; RETURN
1500 Set graphics mode

Lines 1600–1640 set character colours

Line 1600 Calculate byte in colour RAM
1610 If bit 1 set then set foreground 1
1620 If bit 2 set then set background 2
1630/1640 If bit 3 set then set foreground 2; RETURN

'Multi' Program Part Three: Listing

```
2000 OPEN 1,1,1,"COLOR":REM SAVE COLOUR
2010 FOR I=1024*16TO1024*17-1
2020 PRINT#1,PEEK(I)
2030 NEXT I:CLOSE1:RETURN
2100 OPEN 1,1,0,"COLOR":REM LOAD COLOUR
2110 FOR I=1024*16TO1024*17-1
2120 INPUT#1,Q:POKE I,Q
2130 NEXT I:CLOSE1:RETURN
```

```
2200 OPEN 1,1,1,"PICTURE":REM SAVE SCREE
N
2210 CO=0
2220 FORI=1024*24TO1024*32-1
2230 A%=PEEK(I)
2240 IFCO=0ANDNOT(A%=0ORA%=255)THENPRINT
#1,CHR$(A%);:GOTO2295
2250 IFCO=0THENB%=A%:CO=1:GOTO2295
2260 IFA%=B%THENCO=CO+1:GOTO2295
2270 IFB%=0THENCO=-CO
2271 PRINT#1,CHR$(255);CO:CO=0
2280 IFA%=0ORA%=255THENB%=A%:CO=1:GOTO22
95
2290 PRINT#1,CHR$(A%);
2295 NEXTI:IFCO=0GOTO2299
2296 IFB%=0THENCO=-CO
2297 PRINT#1,CHR$(255);CO
2299 CLOSE1:RETURN
2300 OPEN1,1,0,"PICTURE":REM LOAD SCREEN
2310 FORI=1024*24TO1024*32-1
2320 GET#1,A$:IFA$=""GOTO2320
2325 A%=ASC(A$)
2330 IFA%<>255THENPOKEI,A%:GOTO2350
2335 INPUT#1,CO:IFCO<0THENCO=-CO:A%=0
2340 FORJ=1TOCO:POKEI+J-1,A%:NEXTJ:I=I+C
O-1
2350 NEXTI:CLOSE1:RETURN
```

'Multi' Program Part Three: Description

Lines 2000–2350 save and restore screens from cassette. See 'Hires' description for the operation of this section of the program.

'Multi' Program Part Four: Listing

```
3000 BT=0:LF=0:IFF=0ORF=1THENBT=85:LF=1:
REM MULTICOLOUR FILL
3001 IFL=0ORL=1THENBT=BT+170:LF=LF+2
3010 XD=2:XREF=X:YREF=Y
3020 GOSUB4000
3025 GOSUB1600
3030 IFXD=2THENIFX=XHANDPEEK(P)=0THENPOK
EP,BT:X=X+8:GOTO3020
3040 IFXD=-2THENIFX=XH+6ANDPEEK(P)=0THEN
POKEP,BT:X=X-8:GOTO3020
```

```
3055 IFLO<>LF*2↑(XL-1)THENGOSUB1045:X=X+
XD:GOTO3020
3060 XD=-XD:X=XREF:IFXD=-2GOTO3020
3070 X=X+2:Y=Y+1:GOSUB4000
3080 IFLO<>LF*2↑(XL-1)THENGOSUB1045:X=X+
XD:GOTO3020
3090 X=XREF:Y=YREF:RETURN
4000 XH=8*INT(X/8):REM GET A PIXEL
4010 YH=8*INT(Y/8)
4020 XL=7+XH-X
4030 YL=Y-YH
4040 P=24*1024+YL+XH+YH*40
4050 LO=((PEEK(P))AND(2↑XL))OR((PEEK(P))
AND(2↑(XL-1)))
4060 RETURN
5000 SS=SS+1:IFSS/2=INT(SS/2)THEN5050:RE
M SET SPRITE
5005 POKE53271,1:POKE53277,1
5010 POKE53248,XS:POKE53249,SY:POKE53287
,SC:POKE53269,1:POKE53275,1
5020 POKE16384+2040-1024,32:FORI=16384+3
2*64TO16384+33*64:POKEI,255:NEXT
5030 SC=SC+1:IFSC=16THENSC=0
5040 RETURN
5050 POKE53269,0:RETURN
6000 IFSX<0THENSX=0:RETURN:REM MOVE SPRI
TE
6001 IFSX>319THENSX=319:RETURN
6002 IFSY<0THENSY=0:RETURN
6003 IFSY>255THENSY=255:RETURN
6004 IFSX>255THENPOKE53264,1:XS=SX-256
6005 IFSX<256THENPOKE53264,0:XS=SX
6010 POKE53248,XS:POKE53249,SY:RETURN
8000 V=V+1:XT(V)=X:YT(V)=Y:REM TRIANGLE
FILL
8010 IFV<>3THENRETURN
8020 Y1=YT(1):Y2=YT(2):Y3=YT(3):X1=XT(1)
:X2=XT(2):X3=XT(3):V=0
8030 IFY2<Y1THENY2=Y1:X2=X1:Y1=YT(2):X1=
XT(2)
8040 IFY3<Y2THENY3=Y2:Y2=YT(3):X3=X2:X2=
XT(3)
8050 IFY2<Y1THENYT(2)=Y2:XT(2)=X2:Y2=Y1:
X2=X1:Y1=YT(2):X1=XT(2)
```

```
8120 M3=(X3-X1)/(Y3-Y1):C3=Y3*M3-X3:REM
Y3#Y1
8125 SP=1:IFX2>Y2*M3-C3THENSP=-1
8128 IFY2=Y1THEN8158
8129 M1=(X2-X1)/(Y2-Y1):C1=Y1*M1-X1
8130 FORY=Y1TOY2
8140 FORX=2*INT((Y*M1-C1)/2)TO2*INT((Y*M
3-C3)/2)STEPSP*2
8150 GOSUB1000:NEXTX,Y
8158 IFY3=Y2THEN8190
8159 M2=(X3-X2)/(Y3-Y2):C2=Y2*M2-X2
8160 FORY=Y2TOY3
8170 FORX=2*INT((Y*M2-C2)/2)TO2*INT((Y*M
3-C3)/2)STEPSP*2
8180 GOSUB1000:NEXTX,Y
8190 RETURN
```

'Multi' Program Part Four: Description

Lines 3000–3090 perform multicolour fill

Line: 3000 Default byte BT=0
Default bit pair LF=0
If paintbrush mode is plot or invert then BT=64+16+4+1 (all even bits) and LF=1

3001 If level mode is plot or invert then BT=BT+128+32+8+2 (all odd bits) and LF=LF+2

3010 Scan right on alternate X values; save current co-ordinates

3020/3025 Get a pixel; set character colour

3030 If scanning to the right and next byte is zero then set to BT; increment eight pixels; go on to the next byte (start of byte only)

3040 If scanning to the left and next byte is zero then set to BT; decrement 8 pixels; go on to next byte (start of byte only)

3055 If current bit pair (level and mode) not equal to LF shifted by XL-1, then go to plotting routine; increment X; go on to next bit pair

3060 Otherwise we have reached boundary; reverse scan direction; restore X co-ordinate; if scanning to the left go on to next pixel

3070 If scanning to the right increment Y (go onto next row) and X (to avoid duplication); get another bit pair

3080 If bit pair not set, then set it; increment X; go on to next bit pair

	3090	Otherwise we must have scanned down to lower boundary – RETURN
Lines	4000–4060	get a bit pair
	4050	Get odd and even bits
Lines	5000–5050	perform sprite handling
Lines	6000–6010	move the sprite
Lines	8000–8190	handle the triangle fill. See 'Hires' program description. This version is slightly more robust, and it is almost impossible to get an overflow. The inverse of the gradient is used, instead of the gradient of the lines. (Adjust 'Hires' accordingly.) Y3 can never be equal to Y1 unless a fill is attempted on a triangle of zero area. If Y2=Y1 then the top half of the triangle is omitted. If Y3=Y2 then the bottom is omitted. The (inverse) gradient is calculated *within* the appropriate loop only. X is always incremented or decremented by two, and the limits of the loops are always multiples of 2 in the X direction.

'Multi' Program: Controls

All controls are as given for 'Hires', except those noted below.

Clear Screen controls:

X Clears the colour nybble area to light blue as well as colour RAM to black and white

Note that no facility is provided for saving and restoring the colour nybble area.

Parameter controls:

F Paintbrush attribute (bit mapped screen only), range −1 to 2:

−1	Plot pixel in colour 1 for given level (erase)
0	Plot pixel in inverse colour for given level (invert)
1	Plot pixel in colour 2 for given level
2	No change to colour for given level

L Level attribute (bit mapped screen only), range −1 to 2:

−1	Plot pixel in background colour
0	Invert pixel layer
1	Plot pixel in foreground colour
2	No change to layer

Note that both L and F may both be changed at the same time.

C Colour Code (colour RAM and colour nybbles), range 0 to 7:

0	No change to colours while plotting
1	Change foreground colour 1 only
2	Change background colour 2 only
3	Change foreground 1 and background 2 colours
4	Change foreground colour 2 only
5	Change both foreground colours
6	Change background 2 and foreground 2 colours
7	Change all three colours

On depressing the C key the user will be prompted for new values of all four colours, followed by the colour code. The screen colour will change as soon as the final RETURN is entered.

The triangle fill and paint routines have been modified to work as described in the preamble.

Using 'Multi'

Key in the whole program this time, and then RUN it. The copyright notice appears, followed by a screen of randomly coloured blocks. The colour nybble screen clears automatically. Press the X and Z to keys to clear colour RAM and the bit mapped screen. Try drawing a few lines. Notice how vertical lines are much thicker than horizontal ones. To offset this it is often necessary to draw horizontal lines double thickness. (You could try modifying the program to do this automatically.) Screen Graphics 64 uses double size pixels to try and achieve this, but fails owing to the corresponding loss of resolution in the Y direction. This is not a necessary consequence of the method. Notice also how the staircase effect is far more marked on near-vertical lines, than near-horizontal lines, and that the cursor flashes between light blue and white, the two foreground colours. Change mode F to −1, and you can now draw in white over the other colours. Change F to 1 and level L to −1 to draw in black. The four colours, light blue, dark blue, black and white may appear *anywhere* on the screen. Set the sprite to red, and confirm which colours are foreground and which background.

Draw a separate shape in each colour and fill it in. Notice the fill routine works independently of any mode other than the one that it is filling with. Try drawing with L=0 and F=0 to see how the black colour pairs white and

dark blue/light blue swap over. The following table should help you relate the action of the program to what is happening on the screen:

C	Variable	Layer bit	Colour bit	Name	Default colour	Number
0	E	0	0	back1	blue	6
1	B	0	1	back2	black	0
2	A	1	0	front1	white	1
4	D	1	1	front2	light blue	14

Try changing the colours from the default values. Note that for the purposes of the fill routine the values L=0 and F=0 indicate boundary layer=2 and colour=2 respectively, and L=2 and F=2 mean boundary layer=1 and colour=1 respectively, since it isn't really sensible to ask the fill routine to search for a boundary with layer 'invert' or colour 'null'. Also remember that the program is designed such that the background display for a picture should be drawn first followed by the foreground layer.

Despite the precautions taken, the text screen can still interfere with foreground colour 2. Scrolling is inhibited, but every time a character is written to the text screen, the equivalent colour nybble is reset to the current text cursor colour. This does not matter if you call the routines from your own program and avoid using any text. For the package as presented, modify the appropriate routines to restore the colour nybble area to the correct state every time the screen is switched from text to graphics. This is quite easy, but would have confused the program listing unnecessarily.

Calling the 'Multi' routines from your programs

All routines have the same specifications as those of 'Hires' except as noted below.

PLOT: Plots a point. Lines 1000–1070, 1600–1640.
Calls COLOUR routine
Usage: Set X and Y co–ordinates of point
F and L to −1, 0, 1, 2 for erase, invert, plot or null.
C for colour change (0 to 7):
bit 0 change foreground colour 1 (set A)
bit 1 change background colour 2 (set B)
bit 2 change foreground colour 2 (set D)
Returns: P and BY, location of pixel and colour bytes.

CLEAR COLOUR: Clears both colour RAM and colour nybbles.

SET GRAPHICS: Sets multicolour bit mapped mode graphics.

FILL: Paints an enclosed area in current mode and colour attributes up to a boundary defined by pixels of that mode. Lines 3000–4060. Calls FETCH PIXEL and COLOUR routines
Usage: Set A, B, C, D, F and L as above
Call: GOSUB 3000
Returns: As for PLOT and FETCH PIXEL routines

FETCH PIXEL: Returns value of specified pixel as a shifted bit pair in the XLth and XL-1th bits of LO

TRIANGLE FILL: Triangle fill according to A, B, C, D, F and L

Plotting lines and circles

The implementation of these is left as an exercise for the user. Otherwise buy a package such as Screen Graphics 64. You will find that no package ever meets your requirements exactly, so that it is probably better (and cheaper) to implement your own routines. LINE is easy. The triangle fill routine in 'Hires' is sophisticated and inherently contains a line drawing routine which may be modified to draw lines. Circle drawing is a more sophisticated process. The best technique for drawing curves of any kind is Digital Differential Analysis (DDA), but the simplest is to plot points on polar co-ordinates specified by X=Centre+(Radius *COS(A)) and Y-=Centre+(Radius *SIN(A)), as the angle A varies between 0 and 2*PI radius.

There are certain criteria that must be observed when writing incremental routines to draw lines. Firstly, the routines should be efficient (fast), and secondly, straight lines should appear straight. Plotting techniques are fine when drawing lines parallel, or at 45 degrees, to the X and Y axes. Other lines, though they may start and finish at addressable (integer valued) points, may never cross an addressable point in between. Thus we approximate the line by choosing addressable points as close to it as possible. If we choose these well enough the line will appear straight.

Thirdly, lines should start and end in the correct place. Badly designed routines reveal themselves by leaving gaps between lines that should join, or by exhibiting a cumulative error over a large plot. Finally, any lines should have constant density, independent of length and angle. This is probably the most difficult criterion to achieve. Line density is given by the number of pixels turned on along a given section of line. For constant density this number must remain fixed along the length of the line. Normally we just make the line one pixel thick. For straight lines, then, it is necessary to use two related algorithms, one for lines at angles between −45 degrees and 45 degrees from the horizontal, and the other for nearly vertical lines. The weakness in this method shows itself up by the difference in density

between horizontal lines, vertical lines, and lines at 45 degrees. The only cure is to use lines more than one pixel thick. The 'staircasing' effect shown by lines near to the vertical or horizontal is an unavoidable consequence of the limitations of the screen.

The Digital Differential Analyser is a technique which generates curves from their differential equations, which must be known. The equation of a straight line is:

$$\frac{dY}{dX} = \frac{Y2-Y1}{X2-X1}$$

where (X1, Y) and (X2, Y2) are the end points of the line. The simple DDA uses this gradient to simultaneously increment both Y and X by small steps in the ratio of dY/dX. The smallest increment is made equal to one pixel. The integer parts of X and Y are actually used for plotting the points. The truncation involved could cause a cumulative error, so 0.5 is added to the initial values of X and Y to cancel out this effect. The simple DDA generates accurate lines since the displacement of a displayed dot from the true line is never greater than half a pixel. Unlike the more complicated symmetrical DDA, the line produced is always one pixel thick. Much has been written on implementing this algorithm in machine code. Its main weakness is the necessity for an initial division (not very nice in machine code).

Bresenham's Algorithm provides a routine which avoids the initial division. Like the simple DDA, at each iteration the register associated with one of the co-ordinates changes by ± 1. The other register may or may not change depending on the value of an error term maintained by the algorithm, giving the distance, perpendicular to the axis of greatest movement, between the path of the true line, and the actual pixels plotted. Assuming that the error term is measured parallel to the Y axis, and that the X axis is the axis of greatest movement, then at each iteration the slope of the line is added to the error term, E. Before this is done the sign of E is used to determine whether or not to increment the Y register. If E is positive the actual line lies above the current point, so that Y is incremented and 1 is subtracted from E. To remove the necessity for an initial division E is actually scaled, throughout the algorithm, by multiplying it by 2*(X2−X1), since the algorithm is unaffected by multiplying E by a constant.

The following BASIC implementation uses real arithmetic for E and would therefore need modification to be coded efficiently in machine code:

```
10 DX%=X2-X1:DY%=Y2-Y1:X%=X1:Y%=Y1:E=2*D
Y%-DX%
20 FOR I=1 TO DX%
30 PLOT (X%,Y%): REM CALL POINT PLOTTING
40 IF E>0 THEN Y%=Y%+1:E=E+2*DY%-2*DX%
50 IF E<=0 THEN E=E+2*DY%:X%=X%+1
60 NEXT I
```

The differential equation of a circle with its centre at the origin can be written:

$$\frac{dY}{dX} = \frac{-X}{Y}$$

This suggests that we can implement a circular DDA by using −E*X and E*Y as incrementing values:

```
X=X+E*Y
Y=Y-E*X
```

The 'state equations' of the above system can be written:

$$\begin{bmatrix} Y(N+1) \\ X(N+1) \end{bmatrix} = \begin{bmatrix} 1 & E \\ -E & 1-E*E \end{bmatrix} \begin{bmatrix} X(N) \\ Y(N) \end{bmatrix}$$

The state matrix has determinant 1, which means that the system is stable and will draw a circle provided E is small. Setting E to 2↑−N where N is an integer reduces the computation to a pair of shifts and a complement operation. An alternative approach involves the parametric equations of a circle:

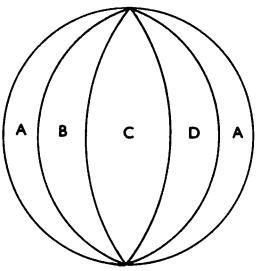
```
X=X*COS(TH)+Y*SIN(TH)
Y=Y*COS(TH)-X*SIN(TH)
```

where the angle TH is kept small. Most circles will not have their centres at the graphics origin, so remember to add a suitable offset to the X and Y co–ordinates when you call the point plotting routine.

Notes

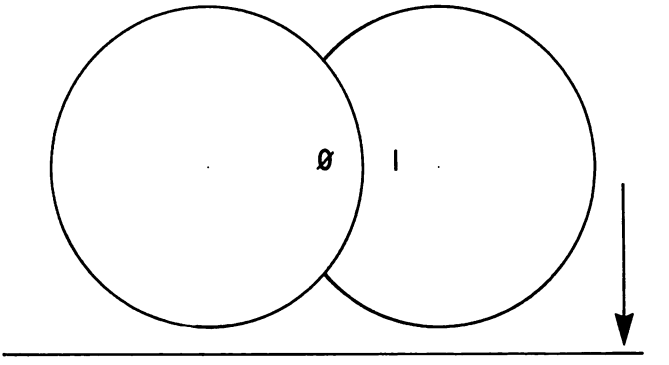
Multicolour mode offers the highest resolution *true* bit mapped colour graphics available on the Commodore 64, 160×200 pixels with 4 colours. Any pixel can be set to any one of four colours (forget the rest), out of the palette of sixteen.

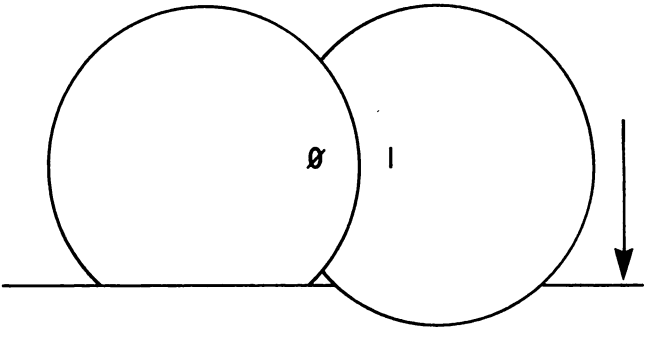
Multicolour mode (particularly text mode) also offers a form of animation often used on machines with large palettes, like the Atari or the BBC Micro. Draw a beachball in multicolour text mode thus:

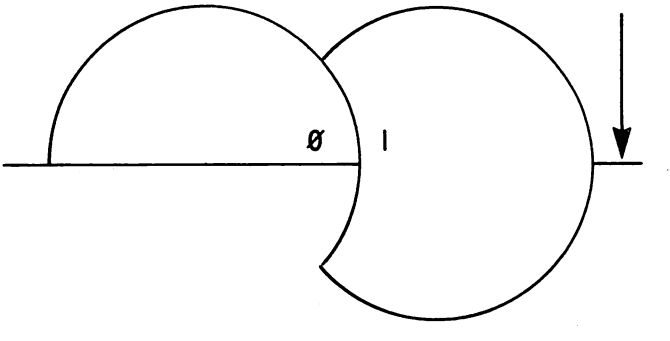


where A, B, C and D are the four possible colours. You will have to design your own characters for this. Now cycle the colour registers in the VIC chip (locations 53281 to 53284) round in a loop, so that A becomes B, B becomes C, C becomes D and D becomes A and so on. The ball appears to spin! This is possibly the simplest form of animation available on the 64.

Finally the reader may have noticed that there is an ambiguity as regards sprite priorities. Sprite 0 is always displayed in front of sprite 1. However if sprite 0 is behind the foreground and sprite 1 is in front of the foreground then there is a problem! According to page 444 of the *Reference Manual* 'MOB vs MOB data is prioritised before priority resolution with character or bit map data'. This is probably the most useless known fact about the Commodore 64, as the results are totally nonphysical, as shown in the diagram overleaf.


Ø
1
foreground


Ø
1
foreground


Ø
1
foreground

7 Animation

In the last chapter we noted a crude animation technique, achieved by changing the definitions of the global colours in multicolour mode. A second method in common use is line animation on a bit mapped screen. This technique is usually only used for demonstration purposes, as it is too slow for anything else. Even rotating a single square on the screen produces a slow and jerky motion. Nevertheless, the Moiré pattern programs that one sees running on computers in shop windows all use this method. The technique is of course simply to leave the computer sitting there drawing, erasing, and redrawing slightly different versions of the same picture. To eliminate the flicker effect (which is caused by the picture spending almost as long in the erased state as it does in the drawn state), and to speed things up, it is possible to set up a number of bit mapped screens in memory (up to a maximum of eight on the 64), each at a different stage in the animation cycle, and then rapidly flick between them. This technique is obviously limited by the low number of bit mapped screens available simultaneously. However, we will use a similar, but far more powerful, technique involving cycling round a large number of sprite frames to achieve smooth rotation of sprites later in this chapter.

Sprites are valuable because, without them, it takes a great many POKEs and well-organised coding to move an object across the screen, erasing it and redrawing it all the while, especially if the object is multicoloured, since one then cannot resort to the trick of drawing only the leading edge, and erasing the trailing edge. Sprites are adequately explained in the *User Manual* (they are one of the few things that are) and further clarified in the *Programmers' Reference Manual*, which contains a fairly decent sprite generator program, so we are not going to duplicate this here. Instead, we open with a section on character graphics – a technique which may have been used to the extent of overkill, but is nevertheless powerful, on machines such as the Sinclair Spectrum and BBC Micro, which lack sprites – and then talk about sprite animation and rotation as preparation for the material on raster interrupts in the next chapter.

Three other subjects which will *not* be dealt with in this book, because they would seem to be adequately covered in the *Reference Manual*, are sprite collision detection, smooth scrolling and joystick/paddle handling. Routines for the latter are given in that book. Suggested exercises in smooth

scrolling are given later in this chapter. After reading the chapter on raster interrupts it is also suggested that the user implements a pool table simulation, or something similar, using interrupt driven sprite collision detection.

Character graphics

Character graphics originate from the days of mainframe games, when moving or static characters on a VDU screen represented a game situation. Later the limitations of the ASCII character set led to extended graphical character sets like those on Commodore's Pet (and the 64). It was soon realised by games writers that, if the character set were held in RAM instead of ROM, a powerful tool for animation was available. If, for instance, the screen is covered with an array of graphical characters looking like little men with their arms in the air, and then the *definition* of the little man character is changed in the character RAM, such that the man has his arms by his sides, then each little man (up to 1000 of them) on the screen will drop his arms to his sides simultaneously. This process takes just eight POKEs. The point is of course that however many little men appear on the screen, only one copy of that man is held in the character RAM, and the screen memory simply contains a collection of 1000 pointers to the relevant character definitions. This is also quite efficient on memory, and it only takes 1K, plus the character RAM (2K), to store a text screen. It takes 8K to store a graphical screen on the 64.

This technique was the only one available for fast screen animation until the advent of sprites. Animation on an ordinary bit mapped high resolution screen is terribly slow because no two things can be achieved simultaneously. Character graphics animation still has many applications today and has been further developed (for instance in the system of pattern definitions on Powertran's Cortex, which allows patterns larger than one character cell to be defined and animated). To see character graphics at its best, play Llamasoft's Matrix (Gridrunner II). At first it may appear that the effects in this game are achieved by superimposing sprites over a bit mapped screen which is being smooth-scrolled downwards. The effect of the background grid moving downwards is actually achieved, however, by defining a series of 'grid segment characters', each at a different stage in the grid's descent, and they are cycled round one by one, giving the effect of smooth scrolling. The game players hurtle round the screen at breakneck speed or appear at the appropriate character position depending on where the rest of the pointers in screen memory point to. The program below simulates this technique in BASIC.

Notice how jerky the movement is, since it takes so long to redefine a character in BASIC and the animation is not synchronised with the raster scan in any way. It is important to ensure that your program is not actually

writing to character memory during the period that the VIC chip is translating the contents of character memory into a signal to send to the television screen. When you have read the following chapter, you could try to implement the following:

Code the program in machine language
Drive it using raster interrupts
Implement the program using smooth scrolling (see *Reference Manual*)
Drive this using raster interrupts

Smooth scrolling is exceptionally jerky and blotchy if driven from BASIC at any speed.

'Char' Program Listing

```
1 REM"■=SAVE"@0:CHAR",8
5 POKE52,48:POKE56,48
10 POKE53281,9
80 POKE53272,(PEEK(53272)AND240)+12
100 FORI=55296TO56295:POKEI,8:NEXT
110 FORJ=0TO24
120 FORI=0TO39:POKE1024+J*40+I,0:NEXT
130 NEXT
170 FORI=0TO7
175 M=255-2↑I
180 FORJ=0TO7
200 POKE12288+J,M
210 IFI=JTHENPOKE12288+J,0
220 NEXT
235 NEXT
237 POKE12288+7,M
240 GOTO170
```

'Char' Description

Line: 5 Reserve memory for user defined characters
Location 52=Bottom of string storage (high byte)
Location 56=Highest address used by BASIC (high byte)
10 Set background colour to orange
80 Move character memory to location 12288 (bits 1–4)
100 Set colour nybble area to brown
110/130 Set every screen location to character 0
170 Loop round the eight possible character frames
175 Set all bits of M except the Ith one

180	Loop round eight bytes of used defined character, redefining it to the next frame of the animation
200	Define vertical lines
210	Define horizontal lines
237	Ensure that no two consecutive rows are set to background colour
240	Repeat

Animated sprites

The Commodore 64 can store up to 1000 blocks of sprite data in memory, but it can only display eight sprites on the screen simultaneously without the use of raster interrupt or flicker techniques. One reason for this apparent inequality is the ability of the 64 to rapidly display each sprite between any of the 1000 sprite frames, thus creating animation. Any sprite can be redefined by any frame in a 256 sprite frame area by just one machine code instruction. In fact there is no point in changing definitions more than fifty times per second, as the television (or your eye) simply could not keep up.

Each sprite consists of an **X** register, a **Y** register, a colour register, and a pointer to a block of sprite data. If the 16K memory bank seen by the VIC chip contains 256 sprite frames, each of 64 bytes, and a sprite is visible on the television screen, then the shape of that sprite pointer for that particular sprite. Remember that the eight sprite pointers are stored in the last eight bytes of the 1024 byte screen memory. Only the first 1000 bytes are actually used for the screen, leaving 24 bytes spare. The other 16 bytes can be used as virtual sprite pointers, as shown in the next chapter. To change the shape of the displayed sprite, all that is necessary is to change the value of the sprite pointer! For rapid animation this should be done using raster interrupts (dealt with in the next chapter) to avoid screen flicker. Using interrupt routines, it is even possible to make animation or rotation occur automatically. Remember also that the last byte in a 64 byte sprite frame is not used (the VIC chip sees the sprite pointer instead) and that the other 63 bytes are used as a miniature bit mapped screen, in a 3*21 byte array, mapping to a 24*21 pixel array (bytes 1 to 3 map to the first row, bytes 4 to 6 the second row, etc.). All this is adequately introduced in the *User's Manual* – and done to death in the *Reference Guide*. All should become clear by following the example given in the next section, which achieves smooth rotation of sprites by cycling the sprite pointers around a block of forty sprite frames. Even three dimensional objects can be made to appear to revolve in space by creating a series of views of a solid object from different angles and swapping the sprite pointer between them. Finally note that, at the other extreme, if every sprite is to look the same then all sprite pointers can be set to indicate the same sprite frame.

Rotating sprites

This subject appears to cause a great deal of confusion. There is no hardware on the Commodore 64 to rotate sprites. It has to be done with software, i.e. sprite animation. One day, sprites may well consist of sets of three dimensional data, and it will be possible to not only move them around the screen, but also to view them from any desired distance or perspective. Until then, we have to rely on generating the data for each view in advance, switching the sprite pointers between the data blocks to create the effect of rotation. This section is about creating such blocks.

A fixed sprite, i.e. one that does not change its shape, but only its orientation on screen, can only be rotated within a 21 pixel diameter circle, and the original design must fit in this circle, as this is the largest area that can be rotated within the 21*24 pixel square sprite area. The most effective method of achieving this is to store the co-ordinates of the ends of a series of straight line segments representing the boundary of the sprite in an array of (X,Y) pairs relative to the centre of the sprite circle, and to then fill this in, if necessary, using a fill routine. That is, if you wish to rotate a hexagon, you will need to create a 6*2 array of co-ordinates (or two 6-vectors). Now decide on the number of frames necessary for a smooth animation. As there are (less than) 21*PI=66 pixels round the circumference of the circle there is no point in creating more frames than this, as some would be identical, and therefore redundant. Note that, if the original object has some degree of symmetry, then the data does not need to be rotated through the full 360 degrees. For instance, if it is desired to create 36 frames at 10 degree intervals, then, in the case of the hexagon (which returns to its original shape when rotated through 60 degrees) only 6 frames need be generated. The program should cycle the sprite pointer through the six frames six times to create the full 360 degree rotation. To create data for the nth frame, the following formulae should be used to generate the co-ordinates of its corners:

$$x' = x\cos(nA) - y\sin(nA)$$
$$y' = x\sin(nA) + y\cos(nA)$$

where x and y give the co-ordinates of a corner, and A is the angle between successive frames. When the rotated frames have been created, use a standard paint routine to fill them in. This is particularly easy if you are using a graphics such as package Screen Graphics 64 and the area to be filled happens to include the centre of the circle. The same technique can be applied in multicolour mode, though the results are not as effective since the largest available circle has a diameter of only 12 pixels, with no convenient central pixel. As the user probably does not possess graphics routines to plot directly to a sprite, the easiest approach is to plot the sprite frames on the ordinary high resolution (multicolour) bit mapped screen, in 21*24 pixel blocks, and then copy them, byte by byte, into the sprite area.

The following routine uses Screen Graphics 64 to create forty sprite frames of a rotating triangle on the high resolution bit mapped screen. It is very easy to translate it for use with Simons' BASIC, any other graphics package, or your own routines, such as those presented in this book.

'Rotate' Program Listing

```
0 N=3
5 HIRES 16,13
10 DIM X(N),Y(N),XD(N),YD(N)
15 DATA 10,1,20,15,1,15
20 FOR I=1 TO N:READ X(I),Y(I):NEXT
30 FOR T=0 TO 3:FOR H=0 TO 9
40 XX=H*32:YY=T*32:TH=(T*90+H*9)*π/180
45 FOR I=1 TO N:XD(I)=(X(I)-10)*COS(TH)-
(Y(I)-10)*SIN(TH)+10
50 YD(I)=(X(I)-10)*SIN(TH)+(Y(I)-10)*COS
(TH)+10:NEXT
55 FOR I=1 TO N-1
60 DRAW XX+XD(I),200-YY-YD(I),XX+XD(I+1)
,200-YY-YD(I+1),12:NEXT
70 DRAW XX+XD(N),200-YY-YD(N),XX+XD((1),
200-YY-YD(1),12
80 FILL XX+10,200-YY-10,12
90 NEXT H,T
100 END
110 FOR T=0 TO 3:FOR H=0 TO 9
115 PRINT 10*T+H
120 FOR A=0 TO 20:FOR B=0 TO 2
130 S=32768+64*(10*T+H)+3*A+B
140 YP=A+T*32:XP=B+H*4
150 YH=INT(YP/8):YL=YP-8*YH
160 P=24576+(YH*40+XP)*8+YL
170 POKE S,PEEK(P)
180 NEXT B,A,H,T
190 END
200 POKE 53248,99:POKE 53249,99:POKE 532
69,01
210 POKE 36865,0
220 END
300 FOR I=0 TO 39
310 POKE 36856,I
320 NEXT I
330 GOTO300
```

The second half of the program copies the sprite frames from the high resolution screen into their correct locations in memory. Unfortunately, if using Screen Graphics 64, this is not easy, as the high resolution screen is stored under the BASIC interpreter. If you have both Screen Graphics 64 (or Ultrabasic), and a machine code monitor such as Supermon (which doesn't clash with Screen Graphics 64) then follow the instructions below *very carefully*. If you do not possess these utilities then read the section on how to modify the program appropriately – and study the listing very carefully.

'Rotate' description

Lines 0–100 create sprite frames on the bit mapped screen

Line 0/20 Switch to hires; set up arrays N=number of points (corners) X(I), Y(I) co-ordinates of the N points

30 T=number of rows of sprites (if more/less than 40 frames are required, then simply change the loop terminator)
H=column counter

40 XX and YY are the offsets for each sprite frame, as presented on the bit mapped screen. Each frame is allocated a 32*32 pixel block, though only the upper left hand side 21*24 block is used. TH is the angle of rotation in radians. This changes by 90 degrees for each row (4), and 9 degrees for each column (10 of them)

45 Cycle through points; Subtract 10 from X and Y co-ordinates to move origin to centre of sprite frame; Use:

$$X' = X\cos(TH) - Y\sin(TH)$$
$$Y' = X\sin(TH) + Y\cos(TH)$$

to calculate rotated position of point; Move origin back again

55 Cycle through points (less one)

60 Join each point to the next one

```
DRAW X1, Y1, X2, Y2, 12
```

joins (X1, Y1) to (X2, Y2) with colour 12

80 Join last point to first one

90 RETURN

Lines 110–190 copy bit mapped screen to sprite area

Line 110 Cycle through rows and columns

115 Print out where the program has got to

120 Cycle through 21*3 sprite bytes

130 Sprite byte=start of sprite area+64*sprite number +3*sprite row+sprite column

140 (XP,YP)=co-ordinate of b.m.m. screen byte

150	High and low bytes of Y co-ordinate
160	Screen byte=start of screen area+Yhigh*40*8+X*8+Ylow
170	Copy screen byte to sprite byte
180	Repeat; Repeat; Repeat
190	Stop

Lines 200–330 perform animation

Line 200/220	Position sprite and make it appear
300/330	Cycle sprite pointer through frames. Note that the sprite pointers start at (16K+1024−8) and the sprite area runs from 17K to (17K+40*64) ($4400–$4E00)

Using 'Rotate'

LOAD machine code monitor
NEW
LOAD Screen Graphics 64 or Ultrabasic
RUN (through initialisation routines)
LOAD "ROTATE"
RUN

Forty triangles should be drawn on the screen and filled in. They will be in four rows of ten and each one will be rotated nine degrees from the previous one. Note that, because the triangle is the same after it has beeen rotated through 120 degrees, only one third of these frames are actually needed, but the routine has been written to produce generalised rotating sprite data. Note that the program must have stopped at line 100. Hit STOP/RESTORE to get back to text mode and then SYS to get your machine code monitor. This is the last time that you should press STOP/RESTORE in the course of these instructions.

We can now copy the bit mapped screen from under the BASIC interpreter ROM, where we cannot do anything with it in BASIC, to the normal locations used in this book. First we must bank out the BASIC ROM. Type:

D 0000

and note that location 0001 is set to 37. Type:

D A000

and note that you can see the start of the BASIC ROM. Now change the contents of location $0001 from 37 to 36 (hexadecimal) and again disassemble from $A000. You can now see the bit mapped screen! Note that it is mainly zeroes (background colour). Now type:

```
T A000 C000 6000
```

and disassemble from $6000 upwards. You have moved the bit mapped screen, and it took less than a second! It now resides in the normal area from decimal location 24K to 32K. Finally change location $0001 back to 37, to enable you to use BASIC, and type X to return to BASIC. All the above commands are Supermon instructions but other monitors such as Supersoft's ZOOM will do the job. Now type:

```
RUN 110
```

and the forty sprite frames will be copied (rather slowly) into the sprite area. This tiny routine, lines 110 to 190, is extremely useful. It provides the basis for a new kind of sprite editor, designed for created data areas for sprite animation purposes, as we will see in the next section. When the program finally reaches sprite 39 (the numbers are printed on the screen), type:

```
RUN 200
```

and a small, black triangular sprite will appear on the screen. Finally, type:

```
RUN 300
```

and the triangle spins round at breakneck speed! Your first sprite animation!

Try slowing it down with a delay loop. Have two or three sprites rotating on the screen at the same time. Make some rotate in opposite directions, at different speeds (put STEP-2 in the FOR . . . NEXT loop at line 300). You can save the sprite area on to tape or disc and restore it at RUN time, rather than recreating it using 'Rotate'. Type S "SPRITES",01,4400,4E00 from Supermon. When you have read the next chapter you will be able to implement raster interrupt-driven rotation and autorotation. For those with Simons' BASIC the same technique as above should be followed except that SB stores the high resolution screen under Kernel, so that it must be moved using:

```
T E000 0000 6000
```

For those without either utility, or without a machine code monitor, or for those who find all the above a bit tedious, implement a line drawing routine based on the triangle fill routine in 'Hires', and then convert the above program to use the 'Hires' routines to draw on the normal ($6000–$8000) bit mapped screen (it's very easy, but slow!). Geniuses could write the whole thing in machine code! Try generating something other than a triangle. Change lines 0 and 15 of 'Rotate' to read:

```
0  N = 12
15 DATA 8,6,6,0,14,0,12,6,20,4,20,12,12,
10,14,20,6,20,8,10,0,12,0,4
```

and go through the whole routine again. You should end up with something that is supposed to be a Maltese Cross, but actually looks more like a demented egg timer spinning at high speed on your screen. This exercise shows the weakness of Screen Graphics 64's fill routine. You should also notice that since some of the frames go fractionally outside the 21 pixel diameter circle the corners of the cross get lopped off.

Save the sprite area, followed by this little program:

```
100 POKE56576,150:POKE53272,8:POKE53265,
59
200 POKE53248,99:POKE53249,99:POKE53269,
255
201 POKE53250,99:POKE53251,150
202 POKE53252,150:POKE53253,99
203 POKE53254,150:POKE53255,150
300 FORI=0 TO 39
310 POKE17400,I:POKE17401,39-I:POKE17402
,39-I:POKE17403,I
320 NEXT I
330 GOTO300
```

Switch the machine off and on again. Load the sprite data, type NEW, load the program and RUN it. Note that if the 'Hires' screen is in use, a maximum of 112 sprite frames can be stored between it and the start of colour RAM.

A sprite editor for animation

First add the program lines given below to both 'Hires' and 'Multi'. You must also choose unused characters to control entry to the routines and add lines saying:

IF A$=" " THEN GOSUB 9000

or whatever – you know the technique. (I used " " and "A".) Subroutine 9000 just draws a set of forty rectangles which will act as borders for the sprite frames. You can use the ordinary screen editors, 'Hires' and 'Multi', to draw sprite frames within these boxes. To simplify the process, subroutine 9110 allows you to copy any one sprite frame to any other on the bit mapped screen. The idea is to draw frame 1, copy it to frame 2, change it slightly, copy this to frame 3 and so on. When you have the frames designed to your satisfaction, use lines 110 onwards of the 'Rotate' program to copy the animation into the sprite area and RUN it.

Sprite Editor Program: Listing

```
9000 FORX7=0TO9:X8=X7*32-1:X9=X7*32+24
9010 FORY7=0TO3:Y8=Y7*32-1:Y9=Y7*32+21
9020 IFX8=-1THENX8=0
9030 IFY8=-1THENY8=0
9040 IFX8=0THEN9060
9050 X=X8:FORY=Y8TOY9:GOSUB1000:NEXT
9060 X=X9:FORY=Y8TOY9:GOSUB1000:NEXT
9070 IFY8=0THEN9090
9080 Y=Y8:FORX=X8TOX9:GOSUB1000:NEXT
9090 Y=Y9:FORX=X8TOX9:GOSUB1000:NEXT
9100 NEXT:NEXT:RETURN
9110 INPUT"FROM X";H8:INPUT" Y";T8
9115 INPUT"  TO X";H9:INPUT" Y";T9
9120 FORA8=0TO20:FORB8=0TO2
9130 YP=A8+T8*32:XP=B8+H8*4
9131 YH=INT(YP/8):YL=YP-8*YH
9132 P8=24576+(YH*40+XP)*8+YL
9140 YP=A8+T9*32:XP=B8+H9*4
9141 YH=INT(YP/8):YL=YP-8*YH
9142 P9=24576+(YH*40+XP)*8+YL
9150 POKE P9,PEEK(P8)
9155 PRINTP8,P9
9160 NEXT B8,A8:RETURN
```

Sprite Editor Program: Description

Lines 9000–9100 draw the boxes

Line 9000/9010 Loop through rows and columns
9020/9040 Don't go off edge of screen
9050/9060 Draw left and right sides
9080/9090 Draw top and bottom of box

Lines 9110–9160 copy a sprite

Line 9110/9115 From where, to where?
9120 Loop through 21*3 bytes
9130/9142 Calculate 'from byte' and 'to byte' (as in 'Rotate')
9150/9160 Copy byte; print sprite co-ordinate; repeat; RETURN

This sprite editor requires a fair bit of practice to use, compared to the more traditional ones based on character arrays (such as that on page 181 of the *Reference Manual*), but enables you to create long, detailed animations at reasonable speeds. The details could, if desired, be tidied up using a sprite editor (such as the Rabbit one which is reviewed in Appendix B).

Experiment with different styles of animation. Try using black and white line drawings – this techinque seems to have been ignored by programmers. Remember, though, that if your cartoon man walks across an object, you will be able to see right through him! Multicolour mode must be used to get round this. Black, white and the three shades of grey also makes for extremely effective animation – especially when something with all the colours of the rainbow suddenly turns up.

Sprite networks

In Chapter 3 we suggested implementing extra functions in the music interpreter, to repeat sections of music, jump to others, etc. This idea could be extended to control animation. Consider implementing a sprite animation of a bird soaring across the sky, diving into the sea, flying slowly over the surface of the water, landing, taking off, etc. Depending on what the program is doing, it might be desired to perform the above in any order, and so a 'network' must be set up round which the sprite pointers can be moved in order to achieve the desired effect at the right time. A simple interpreter, along the lines of 'Music', would be told what to do next, check the network to see where the appropriate sprite frame was located, and then change the sprite pointer accordingly. Thus the network is just a series of sprite pointers, but with repeats and jumps rather like a piece of music! Indeed, a complete animation/music controller could be built in this manner.

8 Raster Interrupts

The picture on your television screen is created by the VIC II chip. Every fiftieth of a second it scans the 64's RAM to decide what the next screen should look like, and sends a message to the modulator in the computer to create the appropriate television signal. For part of this period the signal being created is above the top, or below the bottom, of the normal Commodore 64 screen. During *this* period no information is required from RAM, as the signal for the corresponding lines (or rasters) on the television screen is just 'border colour'. It is highly desirable that all changes made to the television picture, e.g. moving sprites, are made during this period to eliminate screen flicker. We should only make changes to any RAM seen by the VIC chip during the period when it is not accessing it. Consider a sprite moving horizontally across the middle of the screen. If the sprite is moved when the raster scan is halfway down the sprite (it starts at the top and works down, flicking from left to right to create each raster line) then the top half of the sprite would be in the old position, and the bottom half in the new position, which is unlikely to be what was intended!

The above argument is slightly confused by the fact that the VIC chip only creates (just over) 256 rasters down the screen which do not directly correspond to the 625 lines of your television screen. However, in this book we always consider the raster scan with respect to the VIC chip. The visible display window is from raster 51 to raster 251 ($33–$FB).

Sometimes we may wish to make changes to the VIC registers whilst the raster scan is still in the normal screen area, to achieve special effects. These are called split screen effects. For instance, if eight sprites are created in the top half of the screen and, when the raster scan has got to the middle of the screen they are moved into the bottom half of the screen, and then, when the raster gets to the bottom of the screen they are moved back to their original positions, and so on, a total of sixteen sprites will appear on the screen. They will not be flashing on and off (as they would if this was tried from BASIC), because when VIC accesses the RAM corresponding to the top half of the screen the sprites appear to be there, and when it gets to look at the RAM corresponding to the lower half, they are there as well.

There is one other stage in the raster scan sequence when the VIC chip is not accessing RAM apart from the period it spends above or below the normal screen. This is whilst the border to left and right of the screen is being

created. It corresponds to the 'flyback' period of the electron beam in the television tube, and so retains this name. This period is very short and only a small number of machine code operations can be carried out during it. It can be used to make changes (split screen effects) to the screen during the normal raster scan without causing flicker.

All the routines in this chapter may be used simultaneously with any other BASIC or machine code program which does not itself use interrupt routines. Care must be taken when mixing machine code routines that do use interrupts.

Raster interrupts

To synchronise screen changes with the raster scan, or to create split screen effects, we must disable the normal BASIC interrupts and use the VIC chip raster register to create interrupts directly from the raster scan. The interrupt pointers are changed to point to the user interrupt routine, which must include a jump back to the normal interrupt service routine (unless the user is prepared to handle all the CIA chip's functions).

There are a number of important registers connected with this operation (see the table below). To enable an interrupt request the appropriate bit in the interrupt request mask register must be set. The default value of this register is $F0, and for raster interrupts (as opposed to sprite collision or light pen interrupts) bit 0 must be set. If an interrupt occurs, the corresponding bit in the interrupt flag register is set. The appropriate latch (bit) in the interrupt flag register must be cleared before another interrupt (from the same source) can occur. *This is achieved by writing a 1 to that bit.* Writing a 1 to a bit to set it to 0 may seem strange, but anyone who knows about the operation of flip flops will realise why. The BASIC interrupts are disabled by setting the appropriate register in the CIA chips to zero. The raster interrupt register itself ($D012) is a dual function register. It may be interrogated to find the current position of the raster scan. A value written to the raster register (including the MSB) will cause the next raster interrupt to occur, provided all the enables are set correctly, at that particular raster value. For split screen work this must be done, and the appropriate interrupt flag cleared, for every raster interrupt.

Raster Interrupt Routine Useful Locations

Location	Operation
$EA31	Normal interrupt service routine entry
$EA81	Bypass normal interrupt routine return
$D012	Write value for raster interrupt compare (bit 8 is bit 7 of $D011)
$D019	VIC interrupt flag register

$D01A	VIC interrupt mask register
	The bits in the above two locations are as follows:
	7 Any interrupt
	3 Light pen interrupt
	2 Sprite to sprite collision interrupt
	1 Sprite to background collision interrupt
	0 Raster compare interrupt
$DC0E	CIA control register A. Set to:
	00 disables BASIC interrupts
	01 enables BASIC interrupts

Split screen

Our first example of a program utilising raster interrupts allows text and high resolution bit mapped graphics to be displayed on the screen simultaneously. The screen is split horizontally, with a band of graphics across the text screen, or vice versa. The splits can occur at any vertical position down the screen.

The technique is to use the raster interrupt register to create interrupts at two positions down the screen. On the first interrupt we switch to high resolution bit mapped graphics mode, as in Chapter 5. On the second interrupt we switch back to text mode. The routine can be easily modified to switch between other modes. Key in the following program. Monitor/assembler owners can type in the assembly code listing given in the next section directly:

'Split' BASIC Program: Listing

```
1 REM"■=SAVE"@0:SPLIT.B",8
60000 FORI= 49152TO 49296:READA:POKEI,A:
NEXT
60010 DATA 76,16,192,41,1,170,224,0,208,
3,76,49,234,234,234,234,174
60020 DATA 0,221,224,151,208,28,169,150,
141,0,221,169,8,141,24,208
60030 DATA 169,59,141,17,208,169,224,141
,18,208,169,1,141,25,208,76
60040 DATA 129,234,169,151,141,0,221,169
,21,141,24,208,169,27,141,17
60050 DATA 208,169,208,141,18,208,169,1,
141,25,208,76,49,234,255,255
60060 DATA 255,255,169,0,141,14,220,120,
169,0,141,20,3,169,192,141
```

```
60070 DATA 21,3,169,241,141,26,208,88,96
,120,169,49,141,20,3,169,234
60080 DATA 141,21,3,169,240,141,26,208,8
8,169,151,141,0,221,169,21
60090 DATA 141,24,208,169,27,141,17,208,
169,1,141,14,220,96,255
```

Save 'Split.B' to disc or tape, and then change line 60000 to read:

B=0: FOR I=49152 TO 49296: READ A: B=B+A: NEXT: PRINT B

Type RUN and the program should print out 18177. If you get any other value, check your data again carefully. Otherwise, change line 60000 back (or reLOAD the program) and RUN it. Control will be returned to the user. Type:

SYS 49235

and a band of the high resolution screen should appear across the bottom of the text screen. Entering:

SYS 49258

will remove it. Changing locations 49191 and 49219 will change the extent to which the screen is covered with graphics. Both SYS calls and the two POKEs can be made from a BASIC program. After 'Split.B' has been RUN, it can be deleted, as it is stored permanently in memory as a machine code program. All the 'Hires' and 'Multi' routines can be used to plot on the split screen. The text screen scrolls normally.

'Split' Assembler Program: Listing

```
., C000  4C 10 C0 JMP $C010
., C003  EA       NOP
., C004  EA       NOP
., C005  EA       NOP
., C006  EA       NOP
., C007  EA       NOP
., C008  EA       NOP
., C009  EA       NOP
., C00A  EA       NOP
., C00B  EA       NOP
., C00C  EA       NOP
., C00D  EA       NOP
., C00E  EA       NOP
., C00F  EA       NOP
., C010  AE 00 DD LDX $DD00
., C013  E0 97    CPX #$97
```

```
., C015  D0 1C    BNE $C033
., C017  A9 96    LDA #$96
., C019  8D 00 DD STA $DD00
., C01C  A9 08    LDA #$08
., C01E  8D 18 D0 STA $D018
., C021  A9 3B    LDA #$3B
., C023  8D 11 D0 STA $D011
., C026  A9 E0    LDA #$E0
., C028  8D 12 D0 STA $D012
., C02B  A9 01    LDA #$01
., C02D  8D 19 D0 STA $D019
., C030  4C 81 EA JMP $EA81
., C033  A9 97    LDA #$97
., C035  8D 00 DD STA $DD00
., C038  A9 15    LDA #$15
., C03A  8D 18 D0 STA $D018
., C03D  A9 1B    LDA #$1B
., C03F  8D 11 D0 STA $D011
., C042  A9 D0    LDA #$D0
., C044  8D 12 D0 STA $D012
., C047  A9 01    LDA #$01
., C049  8D 19 D0 STA $D019
., C04C  4C 31 EA JMP $EA31
., C04F  FF       ???
., C050  FF       ???
., C051  FF       ???
., C052  FF       ???
., C053  A9 00    LDA #$00
., C055  8D 0E DC STA $DC0E
., C058  78       SEI
., C059  A9 00    LDA #$00
., C05B  8D 14 03 STA $0314
., C05E  A9 C0    LDA #$C0
., C060  8D 15 03 STA $0315
., C063  A9 F1    LDA #$F1
., C065  8D 1A D0 STA $D01A
., C068  58       CLI
., C069  00       BRK
., C06A  78       SEI
., C06B  A9 31    LDA #$31
., C06D  8D 14 03 STA $0314
., C070  A9 EA    LDA #$EA
., C072  8D 15 03 STA $0315
., C075  A9 F0    LDA #$F0
```

```
., C077  8D 1A D0 STA $D01A
., C07A  58       CLI
., C07B  A9 97    LDA #$97
., C07D  8D 00 DD STA $DD00
., C080  A9 15    LDA #$15
., C082  8D 18 D0 STA $D018
., C085  A9 1B    LDA #$1B
., C087  8D 11 D0 STA $D011
., C08A  A9 01    LDA #$01
., C08C  8D 0E DC STA $DC0E
., C08F  00       BRK
```

'Split' Program Description

C000	Jump to start of interrupt routine
C003–C00F	Spare locations for delay loop
C010–C013	Check to see which VIC bank is in use (This tells us whether we are in text or graphics mode)
C015	If graphics mode, go to text routine ($C033)
C017–C023	Switch to graphics
C017–C019	Switch to bank 1
C01C–C01E	Move VIC base addresses
C021–C023	Set bit mapped mode
C026–C028	Set raster compare to #$E0
C02B–C02D	Clear interrupt request register to 'raster'
C030	Restore registers and return to BASIC
C033–C03F	Switch to text
C033–C035	Switch to bank 0
C038–C03A	Move VIC base addresses
C03D–C03F	Set text mode
C042–C044	Set raster compare to #$D0
C047–C049	Clear interrupt request register to 'raster'
C04C	Return to normal interrupt service routines
C053–C069	Switch on split screen
C053–C055	Switch off BASIC interrupts
C058	Disable all interrupts
C059–C060	Change interrupt entry point to $C000
C063–C065	Enable raster compare IRQ
C068	Enable interrupts
C069	Return to monitor (change this to RTS for use from BASIC)

C06A–C085	Switch off split screen
C06A	Disable all interrupts
C06B–C072	Change interrupt entry point to $EA31
C075–C077	Disable raster compare IRQ
C07A	Enable interrupts
C07B–C087	Switch to text mode (see above)
C08A–C08C	Switch on BASIC interrupts
C08F	Return to monitor (RTS from BASIC)

To switch the routine on from your machine code monitor, type:

```
G C053
```

and to switch it off, type:

```
G C06A
```

(or the equivalent instructions). Locations $C027 and $C043 contain the start and end of the textual part of the screen.

We are lucky with the above program, as all switching occurs on the flyback, when the raster is not on the screen. When using this technique within more sophisticated applications, the time delays within the software may cause the switching to occur on the screen, giving a flickery effect. This can be overcome by inserting an extra time delay in locations $C003–$C00F to cause the switching to occur off the screen. This usually causes the screen changes to occur a raster line (or two) further down the screen than intended, easily counteracted by decreasing the raster compare values appropriately.

Under such circumstances it is often desirable to make the switch occur in as short a time interval as possible (very few operations can be carried out in the amount of time it takes to move the raster across the screen). The best way to do this is to reorganise lines C017–C023 and C033–C03F as follows, loading the registers first, and then POKEing the locations all in one go.

Switch to Graphics	Switch to Text
LDA #$96	LDA #$97
LDX #$08	LDX #$15
LDY #$3B	LDY #$1B
STA $DD00	STA $DD00
STX $D018	STX $D018
STY $D011	STY $D011

Try changing the code at C030 to read JMP $EA31. Notice that the text cursor now flashes twice as fast, because the normal interrupt service routines are being called twice as often. The BASIC interrupt routines are normally called fifty times a second, which is very convenient, as this is precisely the

number of times that the television screen is refreshed, so that returning to location $EA31 once per screen refresh gives exactly the same result. The routine can easily be modified to increase the number of splits down the screen. All modes may be mixed. However, normally only one interrupt should return to $EA31, and the rest should finish by returning to $EA81, to restore the **A**, **X**, and **Y** registers and bypassing the normal interrupt service routines.

The most sensible place to make the jump to $EA31 is always at the beginning of the longest section of screen between two splits. Try changing the code at C053 to read:

```
LDA #$01
```

which is the default value of $DC0E in the CIA. The screen split now becomes very flickery due to the fact that we are no longer disabling the normal BASIC interrupts and they now interfere with the raster interrupts at regular intervals.

'Split' – Useful Locations

Function	Decimal	Hex
Split screen on	49235	C053
Split screen off	49258	C06A
Start of text screen	49191	C027
Start of graphics screen	49219	C043
Start of program	49152	C000
End of program	49296	C090

Sixteen sprites

This next raster interrupt routine displays sixteen sprites on the screen simultaneously. It is not particularly useful in practice, except for special applications, but serves as a good introduction to the concept of False or Virtual Video Chips described in this chapter. The routine has the following limitations. The screen is split horizontally into two sections. Only eight sprites may be displayed in each half. (Even using False VIC chips, only eight sprites may be displayed on any horizontal line without using 'flicker' or 'on the fly' techniques.) The screen split may be repositioned vertically. The sprites in the first half are identical to those displayed in the second half. This is simply because there is nowhere to store information about sixteen sprites simultaneously. A method of overcoming this limitation is considered in the next section. The routine can easily be modified using a monitor/assembler, to split the screen into three, four, or more sections (supporting 24, 32 or more sprites). The machine will, however, be slowed down considerably if this is taken to extremes.

Before we look at the routine itself, consider the maximum number of sprites that can be displayed on the screen. In one of Commodore's handouts, they claim a figure of 256, but this appears to be the result of a misunderstanding, since 256 is actually the theoretical maximum number of sprites that the VIC chip can see at once without bank switching. The number of sprites that can be displayed is limited only by the maximum number that can appear in the X and Y directions. Normally the limit in the X direction is eight. Even using very special techniques, there are only 320 points across the screen, and the width of a sprite is 24 pixels, so that the absolute maximum number of complete sprites is INT(320/24) or 13. Similarly, in the Y direction, the maximum number of complete sprites is INT(200/21) or 9, since there are 200 pixels down the normal display area and 21 pixels down a sprite. Thus, the normal maximum number of complete sprites is 8*9=72. Under these circumstances, each block of eight sprites would be limited to a very narrow band across the screen. At the other extreme, if we accept the existence of partial sprites created by raster switching, then up to 200 sprites can appear down the screen, giving a normal maximum of 1600 partial sprites! This would slow operation of the computer down to a virtual halt, if indeed it could be done at all. Type in the following program, save it to tape or disc and then change line 60000 to add the bytes together, rather than POKEing them into memory, (see last section) thus:

60000 B=0: FOR I=49152 TO 49314: READ A: B=B+A: NEXT: PRINT B

The checksum should be 20125.

'Sprite' BASIC Program Listing

```
 1 REM"■=SAVE"@0:SPRITE.B",8
60000 FORI= 49152TO 49314:READA:POKEI,A:
NEXT
60010 DATA 76,16,192,41,1,170,224,0,208,
6,76,49,234,234,96,160,174
60020 DATA 1,208,236,14,192,208,21,169,2
7,141,17,208,169,0,141,18
60030 DATA 208,169,1,141,25,208,173,15,1
92,76,63,192,169,27,141,17
60040 DATA 208,169,128,141,18,208,169,1,
141,25,208,173,14,192,162,0
60050 DATA 157,1,208,232,232,224,16,208,
247,76,49,234,234,169,0,141
60060 DATA 14,220,169,255,141,21,208,173
,14,192,162,0,157,1,208,232
```

```
60070 DATA 232,224,16,208,247,169,48,162
,0,157,0,208,232,232,105,26
60080 DATA 224,16,208,245,120,169,0,141,
20,3,169,192,141,21,3,169
60090 DATA 241,141,26,208,88,96,120,169,
49,141,20,3,169,234,141,21
60100 DATA 3,169,240,141,26,208,88,169,0
,141,21,208,169,1,141,14
60110 DATA 220,96
```

Change line 60000 back to the original version as listed, and RUN it. Then type:

SYS 49230

Sixteen sprites appear on the screen! Their colours and shapes are determined by the contents of memory at powerup (unless you have been playing with sprites in between) but note that the top and bottom rows are identical. The locations of the sprite pointers and frames are given in the table below:

'Sprite' – Useful Locations

Function	Decimal	Hex	Default Value
Split Screen On	49230	C04E	
Split Screen Off	49287	C087	
Start of First Screen	49182	C01E	$00
Start of Second Screen	49203	C033	$80
Start of Program	49152	C000	
End of Program	49314	C0A3	
Sprite Co-ordinates: Y1	49166	C00E	$60
Sprite Co-ordinates: Y2	49167	C00F	$A0
VIC Chip: X0	53248	D000	
X1	53250	D002	
X2	53252	D004	
. . .	. . .	. . .	
Sprite Pointers (start)	2040	07F8	
(end)	2047	07FF	
Sprite Frame 0 data (start)	0	00	
(end)	62	7E	
Sprite Frame 1 data (start)	64	80	
(end)	126	FE	

Try PEEKing the contents of locations 49166 and 49167 and then changing their values *slightly*. The whole first (or second) row moves up or down. Do the same thing with locations 53248, 53250, 53252, etc. Pairs of sprites, one

from the top half of the screen, and one from the bottom, move left or right together. Thus the locations of the sprites are closely tied together. The sprites can be switched off by typing:

```
SYS 49287
```

What is happening is that the interrupt routine is writing a 96 ($60) to *all* the sprite Y co-ordinate registers in the VIC chip each time it receives an interrupt at raster $00, and writing a 160 ($A0) to *all* the Y co-ordinate registers whenever it gets an interrupt at raster 128 ($80). Thus two complete sets of sprites appear, one at the top of the screen, and one at the bottom. But the X co-ordinates of both sets are given by the settings of the X co-ordinates in the VIC chip, whereas the Y co-ordinates are obtained by the interrupt routine from locations 49166 and 49167. POKE in the Y co-ordinates of the sprites in the VIC chip will have no effect at all.

This technique is of restricted use for general purpose applications, though it is an efficient way of reconfiguring the architecture of the machine for a specific purpose. In the next section we keep two entire copies of the VIC chip, POKEing them alternately into the real VIC chip on receipt of a raster interrupt. Thus all sixteen sprites become entirely independent. The method is efficient since it simply consists of a loop sequence containing forty seven each of LDA and STA instructions every raster interrupt, with no conditionals or branches. Try moving the bottom row of sprites up to the centre of the screen, or moving the top row down. Notice that you cannot get all sixteen into the same half of the screen at once.

Flicker

Finally, change location 49203 to 0. The sprites start flickering at high speed (25 times per second). This is because both raster interrupts now occur at raster $00 and we only get *one raster interrupt per screen* (you can't have two at the same time). Thus the position of the sprites simply alternates between consecutive screens. The advantage of this method is that any of the sixteen sprites may appear anywhere on the screen. This technique can even be used from BASIC, to increase the number of sprites still further. Try adding an extra section at the beginning of the interrupt routine which flashes the back colour on and off at the same rate (off=black!). The sprites then appear to be steady! Another method of reducing flicker is given in the next section.

'Sprite' Assembler Program Listing

```
., C000  4C 10 C0 JMP $C010
., C003  EA       NOP
., C004  EA       NOP
., C005  EA       NOP
., C006  EA       NOP
., C007  EA       NOP
., C008  EA       NOP
., C009  EA       NOP
., C00A  EA       NOP
., C00B  EA       NOP
., C00C  EA       NOP
., C00D  EA       NOP
., C00E  69 97    ADC #$97
., C010  AE 01 D0 LDX $D001
., C013  EC 0E C0 CPX $C00E
., C016  D0 15    BNE $C02D
., C018  A9 1B    LDA #$1B
., C01A  8D 11 D0 STA $D011
., C01D  A9 00    LDA #$00
., C01F  8D 12 D0 STA $D012
., C022  A9 01    LDA #$01
., C024  8D 19 D0 STA $D019
., C027  AD 0F C0 LDA $C00F
., C02A  4C 3F C0 JMP $C03F
., C02D  A9 1B    LDA #$1B
., C02F  8D 11 D0 STA $D011
., C032  A9 80    LDA #$80
., C034  8D 12 D0 STA $D012
., C037  A9 01    LDA #$01
., C039  8D 19 D0 STA $D019
., C03C  AD 0E C0 LDA $C00E
., C03F  A2 00    LDX #$00
., C041  9D 01 D0 STA $D001,X
., C044  E8       INX
., C045  E8       INX
., C046  E0 10    CPX #$10
., C048  D0 F7    BNE $C041
., C04A  4C 31 EA JMP $EA31
., C04D  EA       NOP
., C04E  A9 00    LDA #$00
., C050  8D 0E DC STA $DC0E
., C053  A9 FF    LDA #$FF
```

```
., C055  8D 15 D0 STA $D015
., C058  AD 0E C0 LDA $C00E
., C05B  A2 00    LDX #$00
., C05D  9D 01 D0 STA $D001,X
., C060  E8       INX
., C061  E8       INX
., C062  E0 10    CPX #$10
., C064  D0 F7    BNE $C05D
., C066  A9 30    LDA #$30
., C068  A2 00    LDX #$00
., C06A  9D 00 D0 STA $D000,X
., C06D  E8       INX
., C06E  E8       INX
., C06F  69 1A    ADC #$1A
., C071  E0 10    CPX #$10
., C073  D0 F5    BNE $C06A
., C075  78       SEI
., C076  A9 00    LDA #$00
., C078  8D 14 03 STA $0314
., C07B  A9 C0    LDA #$C0
., C07D  8D 15 03 STA $0315
., C080  A9 F1    LDA #$F1
., C082  8D 1A D0 STA $D01A
., C085  58       CLI
., C086  00       BRK
., C087  78       SEI
., C088  A9 31    LDA #$31
., C08A  8D 14 03 STA $0314
., C08D  A9 EA    LDA #$EA
., C08F  8D 15 03 STA $0315
., C092  A9 F0    LDA #$F0
., C094  8D 1A D0 STA $D01A
., C097  58       CLI
., C098  A9 00    LDA #$00
., C09A  8D 15 D0 STA $D015
., C09D  A9 01    LDA #$01
., C09F  8D 0E DC STA $DC0E
., C0A2  00       BRK
.
.
```

'Sprite' Program Description

C000	Jump to C010 (normal interrupt entry)
C003–C00D	Spare locations for delay loop
C00E–C00F	Y co-ordinates of two sprite banks

C010–C016	Load **X** register with current Y co-ordinate of sprite 0. If this is not the same as co-ordinate Y1 then go to second half of interrupt routine (set up Y1)
Locations C018 to C02A	change from top of screen to bottom
C018–C01F	Set next interrupt to occur at raster $00 (including high bit RC8 in $D011)
C022–C024	Clear interrupt request flag register to raster interrupt request
C027	Load **A** with Y2
C02A	Jump round second half of interrupt routine
Locations C02D to C03C	change from bottom of screen to top
C02D–C034	Set next interrupt to occur at $80 (including high bit RC8 in $D011)
C037–C039	Clear interrupt request flag register
C03C	Load **A** with Y2
Locations C03F to C048	store **A** register in all eight sprite Y co-ordinates in the VIC chip
C03F	Load **X** with zero
C041	Store **A** in sprite 0 Y co-ordinate indexed by **X**, i.e. sprite X/2
C044–C045	Add 2 to **X**
C046–C048	If **X** is less than 17 then repeat
C04A	Jump to normal interrupt service routine (note that since this happens twice per screen, the cursor blinks at twice normal speed)
Locations C04E to C086	control initialisation and interrupt handling
C04E–C073	Initialise VIC chip.
C04E	Switch Interrupt Routine in (entry point)
C04E–C050	Switch off BASIC interrupts
C053–C055	Switch sprites on (sprite appear)
C058–C064	Set up Y registers (see C03F–C048)
C066–C073	Set up X registers (see C03F–C048)
C075–C086	Set interrupts to 'Sprite'
C087–C0A2	Set interrupts to normal

For more details on raster interrupt handling see the 'Split' documentation.

False video chips

We will now look at a routine that keeps two copies of the VIC chip in locations $CA00 to $CA0E and $CA30 to $CA5E (47 bytes each). These are known as false or virtual (as opposed to real) video chips. On receipt of a raster interrupt at raster $00 the routine copies the entire first false video chip (FVIC1) into the VIC chip. On receipt of a raster interrupt at raster $80 the routine copies the entire second false video chip (FVIC2) into the chip. Thus, according to the settings of FVIC1 and FVIC2, up to eight sprites may appear in both the top and bottom halves of the screen, and they are all independent with regard to position, colour, mode, size, etc.

POKEing values into the real VIC chip registers will have no effect whatsoever whilst the interrupt routine described above is running, as they are continually being overwritten by the values from FVIC1 and FVIC2. The sixteen (potential) sprites are controlled by POKEing values into the 2*47 bytes in the FVIC1 and FVIC2 locations.

Virtual sprite pointers

The only limitation of the routine as described above is that the sixteen sprites still appear as eight pairs of sprites having the same *shape*. This is because the sprite shapes are obtained by the VIC chip via the sprite pointers at the end of the screen area from the sprite data frames – and there are only eight sprite pointers. To overcome this, two sets of eight virtual sprite pointers, considered as extensions to FVIC1 and FVIC2, are set up in locations $CF00 to $CF07 and $CF10 to $CF17. On receipt of interrupts at rasters $00 and $80 the two virtual sprite pointer sets are copied alternately into the sprite pointer area. Each of the sixteen virtual sprite pointers may be set to point to a different sprite data frame. POKEing values into the real sprite pointer area while the interrupt routine is running will have no effect. Thus we have created sixteen totally independent sprites, with the single limitation that only eight may appear in each half of the split screen. By using the flicker technique – with both interrupts occurring at raster $00 – all sixteen sprites may appear anywhere on the screen.

Zero-page pointers

Because the interrupt routine is continually copying large(ish) amounts of data from one place to another, it is convenient to use the indirect indexed addressing mode of the 6510 chip. The indexing is used to cycle through the locations we are copying. However, we wish to use the same index register (in this case **Y**) for both the addresses we are copying from and those we are copying to. The areas have different base (start) addresses, so we copy indirectly via the spare Zero-page locations at $FB to $FE. The same two pairs of bytes are used for the indirect addressing whether we are transferring the

virtual sprite pointers or the false video chips themselves. Thus in the second case the start address of the VIC chip is held in locations $FE and $FD, and the start address of the false chip is held in $FC and $FB; location $FB will hold the values #$00 or #$30 depending whether we are dealing with FVIC1 or FVIC2. In the first case the start address of the sprite pointers is held in $FE and $FD and the start addresses of the virtual sprite pointers are held in $FC and $FB; location $FB will hold the values #$00 or #$10 depending whether we are dealing with FVIC1 or FVIC2.

'FVIC1' BASIC Program Listing

```
1 REM"■=SAVE"@0:FVIC1.B",8
60000 FORI= 49152TO 49392:READA:POKEI,A:
NEXT
60010 DATA 76,16,192,234,234,234,234,234
,234,234,234,234,234,234,234,234
60020 DATA 234,166,251,224,48,208,5,162,
0,76,29,192,162,48,134,251
60030 DATA 160,0,177,251,145,253,200,192
,47,208,247,76,176,192,234
60040 DATA 234,234,169,0,133,251,169,202
,133,252,169,0,133,253,169
60050 DATA 208,133,254,160,0,177,253,145
,251,200,192,47,208,247,169
60060 DATA 48,133,251,160,0,177,253,145,
251,200,192,47,208,247,169
60070 DATA 0,141,14,220,169,0,141,66,202
,169,128,141,18,202,169,1
60080 DATA 141,25,202,141,73,202,169,27,
141,17,202,141,65,202,169,241
60090 DATA 141,26,202,141,74,202,120,169
,0,141,20,3,169,192,141,21
60100 DATA 3,169,241,141,26,208,88,96,12
0,169,49,141,20,3,169,234
60110 DATA 141,21,3,169,240,141,26,208,8
8,169,0,141,21,208,169,1
60120 DATA 141,14,220,96,64,166,251,224,
48,208,4,169,16,133,251,169
60130 DATA 207,133,252,169,248,133,253,1
69,7,133,254,160,0,177,251
60140 DATA 145,253,200,192,8,208,247,166
,251,224,16,208,4,169,48,133
60150 DATA 251,169,202,133,252,169,0,133
,253,169,208,133,254,224,16
60160 DATA 208,3,76,49,234,76,129,234
```

Using 'FVIC1'

Key in the 'FVIC1' program as above, and save it to tape or disc. Change line 60000 to read:

60000 B=0: FOR I=49152 TO 49392: READ A: B=B+A: NEXT: PRINT B

The checksum should be 35557. Change the line back and RUN the program. Type:

SYS 49200

You now have a computer which has sixteen potential sprites. Type:

POKE 51733, 255: POKE 51781, 255

This enables all sixteen sprites. Now, by POKEing the X and Y position registers in FVIC1 and FVIC2 (locations 51712 to 51727 and 51760 to 51775) with suitable values to move the sprites on screen, they can be made to appear. Remember that the Y values for FVIC1 must be less than 128 and the Y values for FVIC2 must be greater than 128. If you only get 14 sprites it is because two of them are set to the background colour – try moving the BASIC cursor underneath them. To switch the sprite interrupt routine off, type:

SYS 49299

To switch the routine back in without resetting everything, use:

SYS 49242

The following program will make all sprites appear:

```
10 SYS49200
20 POKE 51733,255:POKE 51781,255
30 FOR I=0 TO 15 STEP 2
40 POKE51712+I,30+I*12:POKE 51713+I,100
50 POKE51760+I,30+I*12:POKE 51761+I,200
60 NEXT
```

Flicker

To get the sprites to start flashing we have to set both interrupts to raster zero and alternate the positions of the sprites between consecutive screens. The 'flicker' method has the advantage that any of the sixteen sprites may appear anywhere on the screen, as we shall see in the next section. Type:

POKE 51730,0

to start the process off, and:

POKE 51730,128

to return things to normal. By combining raster interrupts and flicker, 32 sprites can be created with only two interrupts per screen, 16 in each half.

Note that if the grey level of a sprite is the same as the grey level of the background colour, the flicker effect is much less noticeable and a similar result to colour mixing is achieved via a different method. Indeed, using raster interrupts to synchronise the switching of any object between two colours in the same grey level set gives rise to quite a good method of colour mixing.

Note also that the address POKE was not in the program but was the raster interrupt register of FVIC1, which is normally set to 128. The act of copying FVIC1 into VIC is all that is necessary to set up all the registers for the next interrupt.

Virtual sprite pointers

Note that we apparently still have eight pairs of sprites on the screen. Try POKEing (any) numbers into the virtual sprite pointer areas, locations 52992 to 52999 and 53008 to 53015. Things should improve! Type:

```
POKE 52992,128
FOR I=64*128 TO 65*128-2: POKE I,255: NEXT
```

One of the sprites will turn into a solid block of white. If a sprite pointer is set to the value N, then the sprite frame that is displayed is stored in the 63 bytes from (B+64*N) to (B+65*N−2) where B is the Base (start) address of the 16K area that the VIC chip sees (0 in this case).

'FVIC1' Program Assembler Listing

```
., C000  4C 10 C0  JMP $C010
., C003  EA        NOP
., C004  EA        NOP
., C005  EA        NOP
., C006  EA        NOP
., C007  EA        NOP
., C008  EA        NOP
., C009  EA        NOP
., C00A  EA        NOP
., C00B  EA        NOP
., C00C  EA        NOP
., C00D  EA        NOP
., C00E  EA        NOP
., C00F  EA        NOP
., C010  A6 FB     LDX $FB
., C012  E0 30     CPX #$30
., C014  D0 05     BNE $C01B
```

```
., C016  A2 00     LDX #$00
., C018  4C 1D C0  JMP $C01D
., C01B  A2 30     LDX #$30
., C01D  86 FB     STX $FB
., C01F  A0 00     LDY #$00
., C021  B1 FB     LDA ($FB),Y
., C023  91 FD     STA ($FD),Y
., C025  C8        INY
., C026  C0 2F     CPY #$2F
., C028  D0 F7     BNE $C021
., C02A  4C B0 C0  JMP $C0B0
., C02D  EA        NOP
., C02E  EA        NOP
., C02F  EA        NOP
., C030  A9 00     LDA #$00
., C032  85 FB     STA $FB
., C034  A9 CA     LDA #$CA
., C036  85 FC     STA $FC
., C038  A9 00     LDA #$00
., C03A  85 FD     STA $FD
., C03C  A9 D0     LDA #$D0
., C03E  85 FE     STA $FE
., C040  A0 00     LDY #$00
., C042  B1 FD     LDA ($FD),Y
., C044  91 FB     STA ($FB),Y
., C046  C8        INY
., C047  C0 2F     CPY #$2F
., C049  D0 F7     BNE $C042
., C04B  A9 30     LDA #$30
., C04D  85 FB     STA $FB
., C04F  A0 00     LDY #$00
., C051  B1 FD     LDA ($FD),Y
., C053  91 FB     STA ($FB),Y
., C055  C8        INY
., C056  C0 2F     CPY #$2F
., C058  D0 F7     BNE $C051
., C05A  A9 00     LDA #$00
., C05C  8D 0E DC  STA $DC0E
., C05F  A9 00     LDA #$00
., C061  8D 42 CA  STA $CA42
., C064  A9 80     LDA #$80
., C066  8D 12 CA  STA $CA12
., C069  A9 01     LDA #$01
., C06B  8D 19 CA  STA $CA19
```

```
., C06E  8D 49 CA STA $CA49
., C071  A9 1B    LDA #$1B
., C073  8D 11 CA STA $CA11
., C076  8D 41 CA STA $CA41
., C079  A9 F1    LDA #$F1
., C07B  8D 1A CA STA $CA1A
., C07E  8D 4A CA STA $CA4A
., C081  78       SEI
., C082  A9 00    LDA #$00
., C084  8D 14 03 STA $0314
., C087  A9 C0    LDA #$C0
., C089  8D 15 03 STA $0315
., C08C  A9 F1    LDA #$F1
., C08E  8D 1A D0 STA $D01A
., C091  58       CLI
., C092  60       RTS
., C093  78       SEI
., C094  A9 31    LDA #$31
., C096  8D 14 03 STA $0314
., C099  A9 EA    LDA #$EA
., C09B  8D 15 03 STA $0315
., C09E  A9 F0    LDA #$F0
., C0A0  8D 1A D0 STA $D01A
., C0A3  58       CLI
., C0A4  A9 00    LDA #$00
., C0A6  8D 15 D0 STA $D015
., C0A9  A9 01    LDA #$01
., C0AB  8D 0E DC STA $DC0E
., C0AE  60       RTS
., C0AF  40       RTI
., C0B0  A6 FB    LDX $FB
., C0B2  E0 30    CPX #$30
., C0B4  D0 04    BNE $C0BA
., C0B6  A9 10    LDA #$10
., C0B8  85 FB    STA $FB
., C0BA  A9 CF    LDA #$CF
., C0BC  85 FC    STA $FC
., C0BE  A9 F8    LDA #$F8
., C0C0  85 FD    STA $FD
., C0C2  A9 07    LDA #$07
., C0C4  85 FE    STA $FE
., C0C6  A0 00    LDY #$00
., C0C8  B1 FB    LDA ($FB),Y
., C0CA  91 FD    STA ($FD),Y
```

```
.; C0CC  C8        INY
.; C0CD  C0 08     CPY #$08
.; C0CF  D0 F7     BNE $C0C8
.; C0D1  A6 FB     LDX $FB
.; C0D3  E0 10     CPX #$10
.; C0D5  D0 04     BNE $C0DB
.; C0D7  A9 30     LDA #$30
.; C0D9  85 FB     STA $FB
.; C0DB  A9 CA     LDA #$CA
.; C0DD  85 FC     STA $FC
.; C0DF  A9 00     LDA #$00
.; C0E1  85 FD     STA $FD
.; C0E3  A9 D0     LDA #$D0
.; C0E5  85 FE     STA $FE
.; C0E7  E0 10     CPX #$10
.; C0E9  D0 03     BNE $C0EE
.; C0EB  4C 31 EA  JMP $EA31
.; C0EE  4C 81 EA  JMP $EA81
```

'FVIC1' Program Description

C000	Jump to beginning of interrupt routine.
C003–C00F	Spare locations for delay loop.
C010–C01D	Check for FVIC1 or FVIC2:if $FB is set to #$00, set it to #$30; if it is set to #$30, set it to #$00.

Locations C01F–C028 copy current false video chip into VIC

C01F	Zeroise **Y**
C021	Load **A** with FVIC+**Y**
C023	Store **A** in VIC+**Y**
C025	Increment **Y** register (counter)
C026–C028	If **Y** <=47 then repeat
C02A	Go to virtual sprite pointer handling. This instruction may be changed to JMP $EA31 if eight independent sprite pointers is satisfactory (remember to alternate with $EA81 if normal interrupt service routine entry rate is required)

Location C030 is entry point for starting interrupt routine (Initial entry)

C030–C03E	Set up Zero-page pointers (FVIC1)
C040–C049	Copy VIC into FVIC1 (initialisation)
C04B–C04D	Change Zero-page pointers to FVIC2
C04F–C058	Copy VIC into FVIC2 (initialisation)

Location C05A is entry point for restarting Interrupt routine (Re-entry)

C05A–C061	Set up FVIC2 to cause next interrupt at raster #$00
C064–C066	Set up FVIC1 to cause next interrupt as raster #$80 (128)
C069–C06E	Clear FVIC interrupt flag registers
C071–C076	Set up FVIC control registers
C079–C07E	Set up FVIC interrupt request mask
C081–C091	Change interrupt pointers
C08C–C08E	Set up VIC interrupt request mask
C092	BRK or RTS

Location C093 is entry point for stopping Interrupt routine

C093–C0A3	Return interrupts to normal
C0A4–C0A6	Switch off sprites (they're a nuisance!)
C0A9–C0AB	Switch on BASIC interrupts
C0AE	BRK or RTS

Locations C0B0–C0EE control virtual sprite pointer handling

C0B0–C0B8	If $FB=#$30 then change to #$10
C0BA–C0C4	Change Zero-page to sprite pointers (actually they're pointers to pointers)
C0C6–C0CF	Copy virtual sprite pointers into real sprite pointer area
C0D1–C0E5	Return Zero-page pointers to various VIC chips
C0E9–C0EE	Return to normal interrupt service routine if FVIC2, otherwise return to calling program

False Video Chip Locations

VIC Registers No Stores	VIC		FVIC1		FVIC2	
	Hex	Dec	Hex	Dec	Hex	Dec
0 Sprite 0 X position	D000	53248	CA00	51712	CA30	51760
1 Sprite 0 Y position	D001	53249	CA01	51713	CA31	51761
2 Sprite 1 X position	D002	53250	CA02	51714	CA32	51762
3 Sprite 1 Y position	D003	53251	CA03	51715	CA33	51763
4 Sprite 2 X position	D004	53252	CA04	51716	CA34	51764
5 Sprite 2 Y position	D005	53253	CA05	51717	CA35	51765
6 Sprite 3 X position	D006	53254	CA06	51718	CA36	51766
7 Sprite 3 Y position	D007	53255	CA07	51719	CA37	51767
8 Sprite 4 X position	D008	53256	CA08	51720	CA38	51768
9 Sprite 4 Y position	D009	53257	CA09	51721	CA39	51769
10 Sprite 5 X position	D00A	53258	CA0A	51722	CA3A	51770
11 Sprite 5 Y position	D00B	53259	CA0B	51723	CA3B	51771
12 Sprite 6 X position	D00C	53260	CA0C	51724	CA3C	51772
13 Sprite 6 Y position	D00D	53261	CA0D	51725	CA3D	51773
14 Sprite 7 X position	D00E	53262	CA0E	51726	CA3E	51774
15 Sprite 7 Y position	D00F	53263	CA0F	51727	CA3F	51775
16 Sprite X MSB	D010	53264	CA10	51728	CA40	51776
17 Control register	D011	53265	CA11	51729	CA41	51777
18 Raster register	D012	53266	CA12	51730	CA42	51778
19 Light pen X	D013	53267	CA13	51731	CA43	51779
20 Light pen Y	D014	53268	CA14	51732	CA44	51780
21 Sprite appear	D015	53269	CA15	51733	CA45	51781
22 Control register	D016	53270	CA16	51734	CA46	51782
23 Sprite expand Y	D017	53271	CA17	51735	CA47	51783
24 Memory control	D018	53272	CA18	51736	CA48	51784
25 Interrupt flags	D019	53273	CA19	51737	CA49	51785
26 Interrupt request	D01A	53274	CA1A	51738	CA4A	51786
27 Background priority	D01B	53275	CA1B	51739	CA4B	51787
28 Multicolour mode	D01C	53276	CA1C	51740	CA4C	51788
29 Sprite expand X	D01D	53277	CA1D	51741	CA4D	51789
30 Sprite collision	D01E	53278	CA1E	51742	CA4E	51790
31 Background collision	D01F	53279	CA1F	51743	CA4F	51791
32 Border colour	D020	53280	CA20	51744	CA50	51792
33 Background colour 0	D021	53281	CA21	51745	CA51	51793
34 Background colour 1	D022	53282	CA22	51746	CA52	51794
35 Background colour 2	D023	53283	CA23	51747	CA53	51795
36 Background colour 3	D024	53284	CA24	51748	CA54	51796
37 Sprite multicolour 0	D025	53285	CA25	51749	CA55	51797
38 Sprite multicolour 1	D026	53286	CA26	51750	CA56	51798
39 Sprite 0 colour	D027	53287	CA27	51751	CA57	51799
40 Sprite 1 colour	D028	53288	CA28	51752	CA58	51800
41 Sprite 2 colour	D029	53289	CA29	51753	CA59	51801
42 Sprite 3 colour	D02A	53290	CA2A	51754	CA5A	51802
43 Sprite 4 colour	D02B	53291	CA2B	51755	CA5B	51803
44 Sprite 5 colour	D02C	53292	CA2C	51756	CA5C	51804
45 Sprite 6 colour	D02D	53293	CA2D	51757	CA5D	51805
46 Sprite 7 colour	D02E	53294	CA2E	51758	CA5E	51806

FVIC1 – Useful Locations

Function	Decimal	(Hex)
FVIC on (first entry)	49200	C030
FVIC on (re-entry)	49242	C05A
FVIC off	49299	C093
Start of program	49152	C000
End of program	49392	C0F1
Start of upper screen	49248	C060
Start of lower screen	49253	C065

FVIC1 Zero-page Pointers

False VIC chips:	virtual	$FC	$FB
	contents	CA	00 (30)
	real	$FE	$FD
	contents	D0	00
Sprite pointers:	virtual	$FC	$FB
	contents	CF	00 (10)
	real	$FE	$FD
	contents	07	F8

Virtual Sprite Pointer Locations

Sprite pointer	VIC		FVIC1		FVIC2	
	Hex	Dec	Hex	Dec	Hex	Dec
0	07F8	2040	CF00	52992	CF10	53008
1	07F9	2041	CF01	52993	CF11	53009
2	07FA	2042	CF02	52994	CF12	53010
3	07FB	2043	CF03	52995	CF13	53011
4	07FC	2044	CF04	52996	CF14	53012
5	07FD	2045	CF05	52997	CF15	53013
6	07FE	2046	CF06	52998	CF16	53014
7	07FF	2047	CF07	52999	CF17	53015

Dynamic Sprites

We can extend the 'FVIC1' program to give us moving sprites. First type in the following program, and save it:

'FVIC2' BASIC Program Listing

```
1 REM"■=SAVE"@0:FVIC2.B",8
60000 FORI= 49152TO 49453:READA:POKEI,A:
NEXT
60010 DATA 76,0,193,234,234,234,234,234,
234,234,234,234,234,234,234,234
60020 DATA 234,166,251,224,48,208,5,162,
0,76,29,192,162,48,134,251
60030 DATA 160,0,177,251,145,253,200,192
,47,208,247,76,176,192,234
60040 DATA 234,234,169,0,133,251,169,202
,133,252,169,0,133,253,169
60050 DATA 208,133,254,160,0,177,253,145
,251,200,192,47,208,247,169
60060 DATA 48,133,251,160,0,177,253,145,
251,200,192,47,208,247,169
60070 DATA 0,141,14,220,169,0,141,66,202
,169,0,141,18,202,169,1
60080 DATA 141,25,202,141,73,202,169,27,
141,17,202,141,65,202,169,241
60090 DATA 141,26,202,141,74,202,120,169
,0,141,20,3,169,192,141,21
60100 DATA 3,169,241,141,26,208,88,96,12
0,169,49,141,20,3,169,234
60110 DATA 141,21,3,169,240,141,26,208,8
8,169,0,141,21,208,169,1
60120 DATA 141,14,220,96,64,166,251,224,
48,208,4,169,16,133,251,169
60130 DATA 207,133,252,169,248,133,253,1
69,7,133,254,160,0,177,251
60140 DATA 145,253,200,192,8,208,247,166
,251,224,16,208,4,169,48,133
60150 DATA 251,169,202,133,252,169,0,133
,253,169,208,133,254,224,16
60160 DATA 208,3,76,49,234,76,49,234,0,0
,0,0,0,0,0,0,0,0,0,0
60170 DATA 0,0,0,166,251,224,48,208,20,1
62,0,189,0,206,24,125,0,202
60180 DATA 157,0,202,232,224,16,208,241,
76,16,192,162,0,189,16,206,24
60190 DATA 125,48,202,157,48,202,232,224
,16,208,241,76,16,192
```

If you have a monitor/assembler, you could edit the previous assembler program, which is almost identical. The only changes are at lines C000, C064, C0EE and the new routine from C100 to C129 (see below). Change line 60000 to read:

60000 B=0: FOR I= 49152 TO 49453: READ A: B= B+A: NEXT: PRINT B

The checksum should be 41307. Restore the line as it was and RUN the program. Type NEW, and then type in the following program and SAVE it:

```
10 SYS49200
20 POKE 51733,255:POKE 51781,255
30 FOR I=0 TO 15 STEP 2
40 POKE51712+I,30+I*12:POKE 51713+I,100
50 POKE51760+I,30+I*12:POKE 51761+I,200
60 NEXT
70 FOR I=52736 TO 52767
80 POKEI,1:NEXT
```

RUN the program and the screen will be filled with sixteen flickering sprites drifting diagonally down the screen from upper left to lower right. Control is returned to BASIC as usual.

A new routine has been inserted at the beginning of the interrupt routine (actually, for convenience, it is placed at the end and accessed with suitable jumps). Both raster interrupts now occur at raster zero, giving sixteen flickering sprites which may appear anywhere on the screen. The raster interrupt routine first looks in two new blocks of memory, locations $CE00 to $CE0F and $CE10 to $CE1F, called the dynamic sprite velocity areas, and then adds the values found to the current X and Y co-ordinates of the sixteen sprites. Thus we can now not only specify the positions of the sprites, but also their velocities. Refer to the table of dynamic sprite velocity locations given below, and note that we have listed all the X locations followed by the Y locations. Experiment with the values and see how smoothly the sprites move, even at high velocities. You can use the interrupt routine in conjunction with your own BASIC or machine code programs. Note of course that the method works just as well with eight ordinary sprites – I only intend to show how different techniques can be used simultaneously. All the other locations are the same as for the previous routine. Note that, as usual, I have not used the MSB of the X positions of the sprites – this is left to the reader.

Dynamic Sprite Velocity Locations

Sprite		VIC (position)		FVIC1 (velocity)		FVIC2 (velocity)	
Co-ordinate	Number	Hex	Dec	Hex	Dec	Hex	Dec
X	0 (8)	D000	53248	CE00	52736	CE10	52752
X	1 (9)	D002	53250	CE02	52738	CE12	52754
X	2 (10)	D004	53252	CE04	52740	CE14	52756
X	3 (11)	D006	53254	CE06	52742	CE16	52758
X	4 (12)	D008	53256	CE08	52744	CE18	52760
X	5 (13)	D00A	53258	CE0A	52746	CE1A	52762
X	6 (14)	D00C	53260	CE0C	52748	CE1C	52764
X	7 (15)	D00E	53262	CE0E	52750	CE1E	52766
Y	0 (8)	D001	53249	CE01	52737	CE11	52753
Y	1 (9)	D003	53251	CE03	52739	CE13	52755
Y	2 (10)	D005	53253	CE05	52741	CE15	52757
Y	3 (11)	D007	53255	CE07	52743	CE17	52759
Y	4 (12)	D009	53257	CE09	52745	CE19	52761
Y	5 (13)	D00B	53259	CE0B	52747	CE1B	52763
Y	6 (14)	D00D	53261	CE0D	52749	CE1D	52765
Y	7 (15)	D00F	53263	CE0F	52751	CE1F	52767

'FVIC2' Program Assembler Listing

The following changes have been made to the 'FVIC1' program listing:

```
C000          JMP $C100          ; includes new routine
...
C064          LDA #$00           ; flickering sprites
...
C0EE          JMP $EA31          ; interrupt service routine
```

The following new routine has been added at C100:

```
.; C100  A6 FB     LDX $FB
.; C102  E0 30     CPX #$30
.; C104  D0 14     BNE $C11A
.; C106  A2 00     LDX #$00
.; C108  BD 00 CE  LDA $CE00,X
.; C10B  18        CLC
.; C10C  7D 00 CA  ADC $CA00,X
.; C10F  9D 00 CA  STA $CA00,X
.; C112  E8        INX
```

```
., C113  E0 10     CPX #$10
., C115  D0 F1     BNE $C108
.. C117  4C 10 C0  JMP $C010
., C11A  A2 00     LDX #$00
., C11C  BD 10 CE  LDA $CE10,X
., C11F  18        CLC
., C120  7D 30 CA  ADC $CA30,X
., C123  9D 30 CA  STA $CA30,X
., C126  E8        INX
., C127  E0 10     CPX #$10
., C129  D0 F1     BNE $C11C
., C12B  4C 10 C0  JMP $C010
```

'FVIC2' Program Description

C100–C104	Check for FVIC1 or FVIC2
C106–C115	Add velocities to positions
C108	Load **A** with velocity
C10B	Clear carry bit in status register as this is used by ADC
C10C	Add velocity to position
C10F	Update position
C113/C115	All X and Y co-ordinates done?
C117	Return to normal raster routine
C11A–C12B	As above for FVIC2

Other sprite techniques

A second layer of dynamics may be included in the moving sprite system by reserving blocks of 16 bytes to contain the X and Y components of acceleration for each sprite. By adding the acceleration of a sprite to its velocity before updating its position, acceleration is achieved. Using simple second order difference equations (see the notes on DDA given earlier) we can move sprites in circular or sine wave paths without recourse to trigonometry!

By grouping sprites together in pairs which are displayed adjacent to each other on the screen, providing one X and one Y co-ordinate location per pair, and making sure that the raster interrupt routine updates the VIC chip from these locations (e.g. first X register=X, second X register=X+24) double size, normal resolution sprites may be created.

On the fly techniques

We will now discuss a method that causes problems for the Commodore 64. We have talked about increasing the number of sprites by changing the

registers in the VIC chip as the television raster moves *down* the screen. Obviously we will have to perform the same switching on a number of consecutive rasters to, for instance, create more than eight sprites horizontally. As only a vertical scan interrupt is available, precise timing would be essential. Unfortunately, however, though the raster takes around 10 milliseconds to complete one full screen scan, it only takes 26 clock cycles to cross the screen horizontally. As each machine code instruction takes 3 or 4 clock cycles, there is time to do very little in the way of resetting registers during this interval.

The original Atari VCS had no vertical synchronisation – all timing was left to the programmer – and only two sprites (they called them players). To increase the number of players, multiple copies of sprites were not only created down the screen, but across it as well. It was discovered that up to three copies of each sprite could be produced horizontally. This technique was referred to as 'rewrite on the fly'. Bob Whitehead, senior designer and co-founder of Activision, also invented the flicker technique which allows multiple copies of sprites to appear anywhere on the screen. Ironically, this technique was never used by Activision, who considered it to be an unacceptable trade-off. Thus, when a Californian tried to sue because the VCS had a chess piece on its box and Atari did not produce a chess game, Mr. Whitehead had to invent a new technique for getting eight players across the screen. This he called the Venetian Blind method, and it took him two days to complete the display; the chess algorithm was written by Lary Wagner, in conjunction with a national chess champion, and it took him two years.

To display eight chess pieces across the screen (instead of the normal maximum of six – three copies of two players) each object was displayed every other raster line. On each odd scan across the screen, up to four chess pieces were displayed, and on each even scan the other four were shown. The gaps were obvious, but the chess pieces were recognizable. While primitive compared with today's programs, the Atari VCS became a fair chess player. The only problem was that while the VCS calculated the next move, no processor time was left for the display (it would have been impossible to do anything else while maintaining a display like that) and the screen simply displayed random colours.

Try the following from your machine code monitor:

```
  C010    INC $D021      ; increment screen colour
  C013    JMP $C010      ; repeat
G C010
```

A wave of coloured ripples appears down the screen. It can easily be seen that the colour is changing horizontally across the screen as well as vertically down. STOP/RESTORE will return you to BASIC, as the interrupt service routines are still in action. The whole of the rest of the processor power,

however, is involved with changing the screen colour. Try starting the above loop from a raster interrupt so that the ripples no longer drift across the screen. The vertical bands that can be seen represent the end of one machine code operation and the beginning of the next. For BASIC programmers the following program uses raster interrupts and allows you to see the horizontal changes in colour:

```
1 REM"■=SAVE"@0:OTFLY.B",8
60000 FORI= 49152TO 49285:READA:POKEI,A:
NEXT
60010 DATA 76,16,192,41,1,170,224,0,208,
6,76,49,234,234,234,119,173
60020 DATA 15,192,105,128,141,15,192,141
,18,208,169,27,141,17,208,169
60030 DATA 1,141,25,208,162,0,142,33,208
,232,224,249,208,248,76,49
60040 DATA 234,234,234,234,234,234,234,2
34,234,234,234,234,234,234,234
60050 DATA 234,234,234,234,234,255,255,2
55,255,169,0,141,14,220,120
60060 DATA 169,0,141,20,3,169,192,141,21
,3,169,241,141,26,208,88
60070 DATA 96,120,169,49,141,20,3,169,23
4,141,21,3,169,240,141,26
60080 DATA 208,88,169,151,141,0,221,169,
21,141,24,208,169,27,141,17
60090 DATA 208,169,1,141,14,220,96
```

Type SYS 49225 to RUN it. The checksum is 18727. Again go into the machine code monitor. Put sprite 0 into the middle of the screen (set $D000 and $D001 to $80 and $D015 to $FF). Now run this:

```
    CA00    INC $D000
    CA03    JMP $CA00
G   CA00
```

About four diagonally stretched copies of the sprite appear, drifting across the screen. We are again using the entire power of the computer. Try pressing the space bar to delay the interrupt service routine slightly and slow the movement down.

Finally, as an exercise, you could try writing a raster interrupt routine that alternates the position of a sprite between two X co-ordinates, over the vertical raster scan area in which the sprite appears. For further experiments, try the program on page 130 of the *Reference Manual* with different delays in line 80, and also with smooth scrolling in the X direction or diagonally. You will get flickery blotches and black lines appearing across

the screen. This is because every time the video chip smooth scrolls one pixel in any direction it is reinterpreting the way it sees the whole of the screen memory. If this occurs when the current raster position is off the screen it doesn't matter, but if it occurs when the raster is on the screen then the top part of the screen has been interpreted in one way and the lower half in another, causing the flickery effect. Try synchronising this with the raster scan, using raster interrupts, so that scrolling only occurs when the raster is off the screen.

You could also try implementing the pool table simulation suggested in an earlier chapter, using collision interrupts. Use the dynamic sprite routine given earlier. Set interrupt requests for both sprite/sprite collision and raster interrupts at the top of the screen. On receipt of a sprite collision interrupt, check the sprite to sprite collision register at $D01E to see which two (or more) sprites have collided and then simply reverse their directions by subtracting the values of the appropriate dynamic velocity registers from $FF. (For a more physical effect consult a textbook on elementary mechanics!)

Finally, consider what is necessary to move sixteen sprites around a split screen without using flicker. Obviously we can only have eight in each half of the screen but there is no reason why a sprite should not *appear* to pass smoothly across the join by making an identical sprite appear at the right time, in the right place, in the second half of the screen. For instance, consider twelve sprites spread round the screen in a circle, like a clock face, and work out how you could make these twelve sprites rotate smoothly round the screen in a circle.

Implications of false video chips

To summarise, the use of virtual copies of the real video chip and virtual sprite pointers allows us to display more than eight independent sprites on the Commodore 64 screen simultaneously. Using the split screen method each horizontal row is restricted to eight sprites. Using other methods, such as flicker, the sprites may appear anywhere on the screen, but the display becomes unacceptable with more than sixteen sprites. The Venetian Blind, and rewrite on the fly techniques are of very little application on the Commodore 64. The main advantage of using virtual video chips is that all the attributes of each sprite may be set independently. Another advantage of using raster interrupts is that screen updates may be carried out during the 'flyback' period, producing smooth animation, since changes to the video RAM are never carried out during the period when it is being accessed by the video chip. Using extended video chips, dynamic or travelling sprites may be created, with the movement initiated and controlled by an independent machine code or BASIC program, but maintained by the interrupt routine. This is particularly useful with collision detection, and particularly

simple using the flicker method, though the MSB of the X co-ordinate must be set correctly. Automatically rotating or animated sprites may be maintained using a similar technique (updating the sprite pointers from a raster interrupt routine) and double size, or larger, normal resolution sprites can be controlled.

For music and animation, then, the name of the game appears to be interrupt routines, whether used with BASIC or machine code. It is obviously possible to write programs that only use interrupt routines, but except for utilities, this is an inefficient use of the machine. Again, to conserve the power of the machine, it is most economical to restrict fast animation to simple changes, e.g. a dog's legs moving while he is running, and reserve more sophisticated decisions, e.g. what the dog will do next, for changes that occur less often. Only the very simplest changes should be made every screen – preferably only auto-movement/animation and perhaps collision detection. Most animation can be slotted in every fifth or tenth interrupt (i.e. between ten and five decisions/changes per second).

References

(1) *Commodore 64 Programmer's Reference Guide*, Commodore Business Machines Inc., First Edition, First Printing, 1982. (Also Second Edition.)
(2) 'Store the Lit Pixels, Ignore all the Others', Graham Kirby, in *Practical Computing*, November 1982.
(3) *Inside the Commodore 64*, Milton Bathurst, Datacap, Belgium, 1982 edition.
(4) *VIC 1541 Users Manual*, Commodore Business Machines Inc., 1981 edition.
(5) 'Design Case History: The Atari Video Computer System', Tekla S. Perry and Paul Wallich, in *IEEE Spectrum*, March 1983.

Appendix A
Some Useful Locations

Note that there are a number of mistakes in the *Reference Guide*.

Zero-page Memory

$002B–002C	43–44	Pointer to start of BASIC text
$002D–002E	45–46	Pointer to start of variables
$002F–0030	47–48	Pointer to start of arrays
$0031–0032	49–50	Pointer to end of arrays (+1)
$0033–0034	51–52	Pointer to bottom of string storage (moving down)
$0037–0038	55–56	Pointer to limit of memory for BASIC
$0039–003A	57–58	Current BASIC line number
$003B–003C	59–60	Previous BASIC line number
$003D–003E	61–62	Pointer to BASIC statement for CONT
$00B7	183	Length of file name
$00B8	184	Current logical file number
$00B9	185	Current secondary address
$00BA	186	Current device number
$00BB–00BC	187–188	Pointer to file name
$00CS	197	Last key pressed
$00C6	198	Number of characters in keyboard buffer
$00CB	203	Current key pressed: 64 if no key
$00CC	204	Cursor enable: 0=flash cursor
$00CD	205	Cursor timing countdown
$00CE	206	Character under cursor
$00CF	207	Cursor in blink phase
$00DO	208	Input from screen or keyboard
$00D1–00D2	209–210	Pointer to screen line
$00D3	211	Current cursor column
$00D4	212	0=direct cursor, else programmed
$00D5	213	Current screen line length
$00D6	214	Current cursor row
$00FB–00FE	251–254	Free Zero-page space for user programs
$0100–103E	256–511	Processor stack area
$0277–0280	631–640	Keyboard buffer (FIFO)
$0281–0282	641–642	Start of BASIC memory

$0283–0284	643–644	Top of BASIC memory
$0286	646	Current colour code
$0287	647	Colour under cursor
$0288	648	Screen memory page
$0289	649	Maximum size of keyboard buffer
$028A	650	Autorepeat: 128=repeat all keys
$028B	651	Repeat speed counter
$028C	652	Repeat delay counter
$028D	653	Keyboard shift/control flag
$0302–0303	770–771	BASIC warm start link
$030C	780	SYS A register save
$030D	781	SYS X register save
$030E	782	SYS Y register save
$030F	783	SYS status register save
$0310–0312	784–786	USR function jump and address (default B248)
$0314–0315	788–789	Hardware interrupt vector (default EA31)
$0316–0317	790–791	Break interrupt vector (default FE66)
$0318–0319	792–793	NMI interrupt vector (default FE47)
$032E–032F	814–815	Warm start vector (default FE66)

BASIC Interpreter

$A642	NEW
$A65E	CLR
$A69C	LIST
$A742	FOR
$A81D	RESTORE
$A82F	STOP
$A831	END
$A857	CONT
$A871	RUN
$A883	GOSUB
$A8A0	GOTO
$A8D2	RETURN
$A8F8	DATA
$A928	IF
$A93B	REM
$A94B	ON
$A9A5	LET
$AA80	PRINT#
$AA86	CMD
$AAA0	PRINT
$AB7B	GET

$ABA5	INPUT#
$ABBF	INPUT
$AC06	READ
$AD1E	NEXT
$AFE6	OR
$AFE9	AND
$B081	DIM
$B37D	FRE
$B39E	POS
$B3B3	DEF
$B3F4	FN
$B465	STR$
$B6EC	CHR$
$B700	LEFT$
$B72C	RIGHT$
$B737	MID$
$B77C	LEN
$B78B	ASC
$B7AD	VAL
$B80D	PEEK
$B824	POKE
$B82D	WAIT
$B853	subtract
$B947	add
$B9EA	LOG
$BA28	multiply
$BB12	divide
$BC39	SGN
$BC50	ABS
$BCCC	INT
$BF71	SQR
$BFD4	negative
$BFED	EXP
$E097	RND
$E12A	SYS
$E156	SAVE
$E165	VERIFY
$E168	LOAD
$E1BE	OPEN
$E1C7	CLOSE
$E264	COS
$E26B	SIN
$E264	TAN
$E30E	ATN

Kernel Locations

$E37B	Warm restart
$E566	Home cursor
$E5B4	Input from keyboard
$E632	Input from screen
$EA31	Interrupt – clock etc
$EA87	Read keyboard
$EB79	Keyboard select vectors
$EB81	Keyboard 1 – unshifted
$EBC2	Keyboard 2 – shifted
$EC03	Keyboard 3 – Commodore
$EC4F	Set graphics/text mode
$ECB9	Video chip setup
$F20E	Set input device
$F250	Set output device
$F291	Close file
$F30F	Find file
$F31F	Set file values
$F32F	Abort all files
$F333	Restore default I/O
$F34A	Do file open
$F49E	Load program
$F5C1	Print filename
$F5D2	'Loading/verifying'
$F5DD	Save program
$F68F	Print 'saving'
$F6DD	Get time
$F6E4	Set time
$F6ED	Check stop key
$F72D	Find any tape header
$F76A	Write tape header
$F7EA	Find specific header
$F817	'press play'
$F82E	Check tape status
$F838	'press record'
$F841	Initiate tape read
$F864	Initiate tape write
$F875	Common tape code
$F8D0	Check tape stop
$F92C	Read tape bits
$FA60	Store tape characters
$FBC8	Write data to tape
$FBCD	IRQ entry point
$FC57	Write tape leader

$FC93	Restore normal IRQ
$FCB8	Set IRQ vector
$FCCA	Kill tape motor
$FCE2	Power reset entry
$FD9B	IRQ vectors
$FDF9	Save filename data
$FE00	Save file details
$FE25	Read/set top of memory
$FE27	Set top of memory
$FE2D	Set top of memory
$FE34	Read/set bottom of memory
$FE43	NMI entry
$FE66	Warm start
$FEB6	Reset IRQ & exit
$FEBC	Interrupt exit
$FF48	IRQ entry

Appendix B
Utility Software and Book Reviews

SYNTHY64
Abacus (Adamsoft)

Synthy64 is a BASIC-like music orientated interpretive language for the Commodore 64 that avoids all the PEEKs and POKEs necessary to drive the 64's excellent sound synthesiser from BASIC V2.0. The software appears to complement Abacus' Screen Graphics 64, making the powerful hardware in the 64 immediately accessible to the user. There is one major reservation about the implementation of Synthy64, as, unlike Screen Graphics, which adds a number of commands to BASIC, Synthy64 completely replaces BASIC and as a result you cannot do anything else – such as graphics – whilst using the system. Only GOTO and GOSUB are allowed from the normal BASIC instruction set. However, as a stand alone system it is excellent.

A Synthy64 program uses line numbers and the simplest possible music notation. The following program:

```
10 G:A:B:C:D:E:F#:G
```

plays up the scale of G. All eight octaves in three voices are available with sharps and flats, three waveforms, key and tempo change and portamento. Full envelope control over attack, decay and release rates and sustain level is available through the use of easy instructions. Square brackets are used to indicate that a section should be repeated a number of times. Quarter notes are indicated by C/4 and half notes by C/2, etc. You can invent your own instruments by filtering, using synchronisation (for tremolo or vibrato) or ring modulation, or use one of the standard set provided. Music programs can be saved or loaded in the usual way. Synthy64 also has its own set of error messages. It would have been nice to see your program appear on a musical stave on the screen and nicer still if all those music commands were additions to normal BASIC, but for your money (seven pounds or so) you get a cassette with the interpreter, plus three three-voice example programs, and a 44-page manual, which represents excellent value for money.

PETSPEED 64
(Commodore)

Petspeed 64 is Commodore's optimising compiler for the Commodore 64. Actually it is not a compiler at all (by my definition) but a translator to a language called Speedcode. The final translated program includes a fast Speedcode interpreter (a similar idea to Forth). Commodore put no restrictions on the distribution of 'compiled' programs (which are copyable), however to use Petspeed a 'dongle' has to be inserted into joystick port 2. This is not necessary to run a compiled program.

The Speedcode interpreter appears to run around ten times faster than the BASIC interpreter on a translated program. A true compiled program would be in the order of one hundred times faster. This is however a useful increase in speed. The approach Commodore have adopted has the advantage that the resultant Speedcode, including the interpreter, is only about three times larger than the original BASIC program. A compiled program would tend to be many times larger.

Petspeed 64 will translate any clean BASIC V2 program for the 64 into Speedcode provided it is not too large, with minor restrictions on array bounds and even a couple of extensions to the language. It will not compile programs written using such extensions as Simons' BASIC. It is extremely easy to use, and is backed up by a good manual and error trapping facilities. The only chore is making up the initial backup discs required – which is an excellent idea anyway. It is available only on disc. The only problem I had was caused by the Pet DOS corrupting a disc when it became full. (Never fill a 1541 disc.) There were a couple of minor bugs in my review copy which I am assured have now been cured. An excellent utility, if a little expensive.

COMMODORE 64 PROGRAMMER'S REFERENCE GUIDE
(Commodore)

The *Commodore 64 Users Manual* supplied with the computer is 166 pages long and has over 100 mistakes in it. The *Programmer's Reference Manual* is 500 pages long and is one of the best pieces of computer documentation I have ever seen. My main criticism is that it should have come with the computer. Surprisingly enough the two manuals were written by the same person – the first in six weeks and the latter in six months.

The first 100 pages of the *Reference Guide* consist of a complete guide to the resident BASIC interpreter. The author then devotes 77 pages to graphics, 30 pages to sound, 111 pages to machine language, 34 pages to input/output and 90 pages of tables including a fold-out circuit diagram.

The sections on BASIC should take the beginner up to the level where he or she is competent to implement most programs not utilising the special hardware in the machine. Graphics and sound are covered in depth with bit by bit explanations of the registers used to drive the appropriate chips.

Example programs are given at each stage which greatly clarify the text. Program independent POKEs are given for each of the facilities available. Little emphasis is put on how to make creative use of these facilities, however this is not the purpose of the book. Rather too much emphasis is given to the use of sprites at the expense of the five extremely powerful graphics modes and the sound synthesiser. For instance only half a page is devoted to ring modulation, a subject worthy of a chapter itself. The longest chapter in the book is an excellent introduction to machine language programming, with special emphasis on interfacing to the Commodore operating system. Little is said, though, about driving graphics and sound from machine code or the use of of interrupts (hence this book).

The review copy was the second edition of the *Reference Guide* which is printed on high quality glossy paper in two colours and now comes in a spiral bound form which is marginally better than the old plastic binding. Most of the few mistakes in the first edition have been rectified, the price has been reduced in Commodore's usual arbitrary fashion, and the book is now excellent value. The criticisms above were the only ones I could think of and the *Programmer's Reference Guide* is both highly recommended, and necessary for the appreciation of this work.

POWER 64
ProLine (Kobra)

Power 64 helps remove the tedium from all those old fashioned BASIC V2 direct commands that Commodore insisted on putting in the Commodore 64, and adds the ones that should have been there instead. Ironically it is now rather old fashioned itself, having been superceded by packages such as Sysres, Victree and Master64 which are rather cheaper.

It consists of two programs, Power 64 which extends BASIC with commands such as AUTO, DELETE, RENUMBER, SEARCH and REPLACE for editing, and WHY, DUMP and TRACE for debugging. It also provides quick Spectrum-like keystrokes for BASIC, e.g. shift–L for LIST, which is a bit redundant when the 64 already has a similar system (L shift–I for LIST). However you can switch this facility off. TEST allows you to have two BASIC programs in memory at once. A sophisticated facility is also provided to define your own function keys. The definitions are included in the BASIC program as REM statements and so can be saved with it. However, the single most useful facility of Power 64 is probably its ability to scroll BASIC programs downwards as well as up, by holding down the appropriate cursor key. This gives the Commodore 64 one of the most powerful screen editors available.

The second program, Morepower, may be run simultaneously with

Power 64 and adds simplified disc handling commands to the 64. To delete a file compare:

OPEN 15, 8, 15, "S0:filename":CLOSE 15

in BASIC V2 and:

DISK "S0:filename"

in Morepower. Hitting a function key to get a disc directory that does not destroy your current BASIC program is a boon, and the EXEC command which passes control of the computer to a sequential disc file is excellent. Other commands are HEX, MERGE and UNDO (OLD) and there are extended RUN and LIST facilities.

The package comes on a disc from which backup copies cannot be made and is supplied with a rather pretentious manual by Jim Butterworth. Power 64 costs £39 or so. It is something of an anomaly in that it could do with a serious update. A toolkit of this nature is essential to the commercial user, but even the new price should drop significantly before it can become a standard facility for the average home user.

SCREEN GRAPHICS 64
Abacus (Adamsoft)

Abacus Software comes nicely packaged in a resealable polythene bag containing a manual and cassette. In the case of Screen Graphics 64 the manual is a 24 page affair with a colour card cover. The package 'adds more than 20 graphics commands to Commodore 64 BASIC including sprites, high resolution and multicolour graphics'. The cassette contains the machine language routines, a long demonstration program and an effective tutor, written in Screen Graphics 64 BASIC.

The new commands include HIRES for graphics and MULTI to switch to multicolour mode, a unique graphics mode that allows four colours in each eight by eight pixel character square. Unfortunately Abacus have hauled the resolution in the normal multicolour mode to 160 × 100. This is to make the X and Y resolutions equal, but results in rather chunky graphics when, say, drawing a circle. NORM switches to text and GRAPH returns (non-destructively) to graphics.

In both modes TIC plots axis tick marks around the screen, DOT plots a single point and DRAW draws a line, BOX a rectangle, CIRCLE a circle and CHAR writes text, all in any colour. MODE switches between normal, erase and inverse plotting. The remarkably general FILL routine paints solid colour into the most ridiculous shapes, though I did manage to trick it with a simple crescent. PIXEL is a function that returns the state (set or unset) of any point on the screen. BLOCK draws a solid rectangle in the background colour.

DUMP saves your high resolution masterpieces to cassette or disk and GREAD reads them back in. The most complicated picture I could set up took about five minutes to save to cassette, simpler pictures appear to take correspondingly less time.

Screen Graphics 64 adds the following sprite commands to BASIC. COPY creates a sprite data set from data in an appropriately numbered command line as described below. SPRITE turns on a sprite (this command has nine parameters), OFF turns it off again and PLACE positions it at the appropriate place on the screen. There are no commands for raster interrupts, smooth scrolling or collision detection. BIT, COLORS, HEX and SDATA are four commands of a similar nature to the BASIC DATA statement. COPY can read sprite data from either a set of BIT statements, making the program easy to read albeit long, or COLORS in multicolour mode, HEX statements, giving the most compact program, or SDATA statements in the normal DATA format. Up to sixteen data sets can be created though only eight sprites can appear on the screen at once in BASIC.

The function keys f5 and f7 switch the display between text and graphics mode even when a program is running (a stroke of genius), though any graphics command will always switch you back to graphics. In multicolour mode a system of 'paintbrushes' allows logical use of colour. Otherwise all commands allow expressions for parameters.

The graphics demonstration on the cassette includes a sprite cartoon with animated trees, a multicolour house and a spaceman who apparently was desperate to go to the loo. There are three rather slow but impressive 3D plots. The graphics tutor takes you step by step through each graphic command, shows you an example of its use, then runs it on the screen. The manual is detailed, easy to follow and I found no errors in it. My only criticism of the package is that it makes rather limited use of the 64's abilities. It would have been nice to have the option of full resolution in multicolour mode, allow mixing of high resolution and multicolour mode and to have some means of collision detection for sprites. I must say though that having SG–64 eliminates all the frustration due to lack of graphics commands on the 64. I find that it is no hassle to load the tape before a graphics session and am beginning to realise that there are actually advantages to *not* having the firmware built in. Screen Graphics 64 is excellent value for money.

GRAPHICS EDITOR
Rabbit Software

Rabbit software comes packaged in a large display box but without a card insert for the cassette case, though the box insert can be cut to size. Graphics Editor includes a three page instruction sheet. The package consists of a sprite editor and a user defined character generator, allowing interactive

creation and editing of sprite and character data using a simple screen editor which is a delight to use.

Entering the sprite editor one sees three representations of sprite 0: a matrix of 24 by 21 characters with an asterisk representing a set pixel and a space for an unset pixel, and at the right hand side a double size and a normal copy of the actual sprite. It would be nice if double height, single width (and vice versa) copies were also available. The character array can be edited simply by typing a star or a blank over the existing character, causing the two sprites to change simultaneously. The cursor can be moved up, down, left or right. Autorepeat is set on all keys (a nice touch). An apparently unlimited number of sets of sprite data can be created. Simple commands allow the user to switch between sprites, change between high resolution and multicolour modes, change foreground, background and sprite colours and invert the colours. The sprites can also be superimposed, swapped or copied. These utilities will take the drudgery out of setting up data for animated sprites. Sprites can be printed out as a numeric data set suitable for inclusion in a BASIC program, or saved to tape to be restored later. It is a pity that multicolour mode was not treated in more detail, say by having a different character for each colour available thus halving the width of the character array, but as the program appears to be written mainly in BASIC, modification should be easily possible.

In the character editor the user is presented with an 8 by 8 character matrix representing the 64 pixels in the character to be edited, an 8 by 8 display window (no apparent reason why this was not bigger) in which copies of any mixture of characters in the current set may be displayed, thus allowing him to see his character graphics in use, plus a complete copy of the current character set. Editing either the character matrix or the display window involves essentially the same set of commands that the sprite editor uses. There is also a command for colouring all occurrences of a specified character with a particular colour.

I found no bugs in the program and the only time I got locked out was when I accidentally asked for a copy to disc. Typing STOP/RESTORE loses the program. The documentation is a bit confusing as to using the sprites or characters you have generated, but this is a hard subject to explain in a short amount of space anyway, and careful scrutiny of the *Programmer's Reference Manual* should clear up any problems.

EASY SCRIPT
(Commodore)

Easyscript is Commodore's disc or cartridge wordprocessor, and a very good one it is too. It bears a remarkable resemblance to Prime Computer's Runoff, but with a screen editor. As with Power 64, cursor control allows you to scroll the screen both up and down, making it child's play to enter

your text. To overcome the problem of only having a forty column screen you can also scroll it sideways, up to 240 columns. Processing is not carried out as you type stuff in (this does not bother me) but is very quick when you want to see what your output looks like. The output screen can be scrolled left, right and downwards. Easy Script commands are interspersed with the source text and indicated by a reverse asterisk in column one. Disc commands are exhaustive with merge and linking of files, directory, delete etc. available, and the manual contains full instructions for use of the package with a variety of printers.

The top line of the screen indicates the current mode, column and line position. Mode controls are preceded by a function key and allow sophisticated control of the screen, panning, delete, move and insert text, search, tabs, underline, enhance and a soft hyphen. Format commands include justification, centering, alignment, margins, offsets, forced page, pitch, headers, footers, page, and numbering. Special facilities are provided for mail merge, personalised letters and filling forms.

The whole package comes in a presentation box with a manual, crib card and, for discs, two discs, one for backup. The manual is large, thorough, well produced and well written. It contains full training and reference sections and I didn't find any mistakes, though it is sometimes a little difficult to find your way around. Easy Script is rather expensive – these things always are – and Commodore would be well advised to carry their price cutting policy over to their software, but it is an excellent and highly recommended system.

ULTRABASIC
Abacus (Adamsoft)

Ultrabasic is an extended version of Abacus Software's Screen Graphics 64. It is in my opinion the best software library available for the Commodore 64. It is upwards compatible with Screen Graphics, including all its high resolution, multicolour and sprite facilities, but adding turtle graphics and game and sound features, including interrupt driven music, split screen and sprite collision detection. Its main limitation is that only half the normal vertical screen resolution is available in multicolour mode.

Screen Graphics 64 is reviewed above, so only the extensions will be covered here. Turtle graphics allows the user to draw a line by giving a length and a direction from the current point, rather than the co-ordinates of its ends. The turtle graphics commands are pretty standard except that a little turtle shaped sprite actually moves around the screen drawing the line when the commands are executed. Fortunately this can be switched off. The game functions allow the user to read input from joysticks, paddles or a lightpen, access the system timers and detect sprite collisions. Sound commands are available for playing a note and changing its characteristics,

using all of the hardware available in the Commodore 64. A facility is also provided for storing a tune, albeit rather crudely, in a data statement, in a similar way to sprite data. This can then be played using an interrupt driven interpreter so that music can be played at the same time as the rest of your game, or whatever, is running, without disturbing the action. Facilities are also provided for splitting the screen, at any vertical co-ordinates, between text and graphics, implicit repeat (this is a hangover from Synthy 64), and dumping the graphics screen to a CBM or Epson graphics dot matrix printer.

The package includes a 44 page manual with a colour card cover and a disc or tape with Ultrabasic, a demonstration program and a long and detailed tutorial.

GRAPHIX 64
(Supersoft)

Graphix 64 is Supersoft's machine code graphics library for use with 64 BASIC. It is a very powerful and general package but is remarkably lacking in some facilities. You can specify where you want the high resolution screen to go, or have more than one screen simultaneously, but there are no routines for drawing circles or sprites and the package does not support multicolour mode!

Unlike, say, Abacus' Screen Graphics 64, Graphix 64 does not add commands to BASIC, but just provides a set of machine code routines to SYS to. However facilities are provided for parameter passing, in that the SYS command allows multiple parameters. This system allows compatibility with other utilities. An optional wedge is available, if you are not using other utilities, to abbreviate the commands. A split screen option is also available which allows parts of the text and graphics screens to be displayed simultaneously. This will interfere with any other package that uses interrupts.

The other facilities available are pretty standard: switch to graphics or text, clear screen, dotted line, test pixel, set background and foreground colours, set ink and paper colours, set border colour, invert screen, transfer text screen to graphics screen (nice one), area fill or erase, draw or erase or invert line or point, plot text, text window, but no circle!

Graphix 64 comes on cassette with a clear twenty page manual. It is highly recommended for anyone who particularly needs the more powerful facilities provided, but it would probably be better to stick to a more conventional package for general purpose plotting.

ZOOM
(Supersoft)

Supersoft's Zoom looks rather like a souped up version of Jim Butterworth's Supermon. It is an excellent monitor, assembler, disassembler with a lot of extras (indeed there is a command for almost every different letter of the alphabet). It can be easily relocated if necessary.

The commands available are: assemble text, assemble, set break point, compare memory and print differences, disassemble, examine memory (locate), fill memory, go run, hunt for ASCII string, interrogate memory – prints bytes and ASCII strings, load from disc or cassette, jump to subroutine, set mask for locate, relocate code, printer on and off, quick trace, register display, save to cassette or disc, transfer memory, verify cassette or disc, single step, exit and save in Pet format.

Altering many of the screen displays alters the equivalent bytes in memory. There is also a hex to decimal and decimal to hex converter. And a set of disc commands thus: read current status message from the disc drive, display directory, directory with wild cards, validate, change file name, delete with wild cards, reformat, copy between drives with wild cards in file names, full disc backup and display start and end addresses of files on a disc. An excellent and highly recommended package.

INSIDE THE COMMODORE 64
Milton Bathurst (Datacap)

This book, from Belgium, is distributed by Supersoft, and contains complete labelled and fully commented disassemblies of the 64's BASIC and Kernel operating system ROM. It is in two halves, one for each ROM, with complete cross references of all the labels and names at the end of each half.

I have exactly two criticisms. One, nowhere could I find a listing of the actual physical default values of the contents of Page zero memory and, two, there is absolutely no explanation of the contents of the book which, unless you happen to have used a disassembler of the complexity Mr Bathurst obviously used, makes life very confusing. Otherwise this book is absolutely essential to anyone who wants to do any serious work on the Commodore 64, and does not possess a printer, disassembler and full memory map of their own.

Appendix C Hexadecimal/Decimal Converter

Hexadecimal to Decimal

Split the hexadecimal number into high byte and low byte. Look up the decimal values of these under MSB and LSB respectively, and add the results together.

Decimal to Hexadecimal

Find the nearest number below the decimal number in the columns headed MSB. This gives the high byte of the hexadecimal number. Subtract this from the decimal number and look the result up in the columns marked LSB. This gives the low byte of the hexadecimal number.

HEX TO DECIMAL/DECIMAL TO HEX CONVERTER

Hex	LSB	MSB	Hex	LSB	MSB	Hex	LSB	MSB	Hex	LSB	MSB
$0	0	0	$40	64	16384	$80	128	32768	$C0	192	49152
$1	1	256	$41	65	16640	$81	129	33024	$C1	193	49408
$2	2	512	$42	66	16896	$82	130	33280	$C2	194	49664
$3	3	768	$43	67	17152	$83	131	33536	$C3	195	49920
$4	4	1024	$44	68	17408	$84	132	33792	$C4	196	50176
$5	5	1280	$45	69	17664	$85	133	34048	$C5	197	50432
$6	6	1536	$46	70	17920	$86	134	34304	$C6	198	50688
$7	7	1792	$47	71	18176	$87	135	34560	$C7	199	50944
$8	8	2048	$48	72	18432	$88	136	34816	$C8	200	51200
$9	9	2304	$49	73	18688	$89	137	35072	$C9	201	51456
$A	10	2560	$4A	74	18944	$8A	138	35328	$CA	202	51712
$B	11	2816	$4B	75	19200	$8B	139	35584	$CB	203	51968
$C	12	3072	$4C	76	19456	$8C	140	35840	$CC	204	52224
$D	13	3328	$4D	77	19712	$8D	141	36096	$CD	205	52480
$E	14	3584	$4E	78	19968	$8E	142	36352	$CE	206	52736
$F	15	3840	$4F	79	20224	$8F	143	36608	$CF	207	52992
$10	16	4096	$50	80	20480	$90	144	36864	$D0	208	53248

Hex	LSB	MSB	Hex	LSB	MSB	Hex	LSB	MSB	Hex	LSB	MSB
$11	17	4352	$51	81	20736	$91	145	37120	$D1	209	53504
$12	18	4608	$52	82	20992	$92	146	37376	$D2	210	53760
$13	19	4864	$53	83	21248	$93	147	37632	$D3	211	54016
$14	20	5120	$54	84	21504	$94	148	37888	$D4	212	54272
$15	21	5376	$55	85	21760	$95	149	38144	$D5	213	54528
$16	22	5632	$56	86	22016	$96	150	38400	$D6	214	54784
$17	23	5888	$57	87	22272	$97	151	38656	$D7	215	55040
$18	24	6144	$58	88	22528	$98	152	38912	$D8	216	55296
$19	25	6400	$59	89	22784	$99	153	39168	$D9	217	55552
$1A	26	6656	$5A	90	23040	$9A	154	39424	$DA	218	55808
$1B	27	6912	$5B	91	23296	$9B	155	39680	$DB	219	56064
$1C	28	7168	$5C	92	23552	$9C	156	39936	$DC	220	56320
$1D	29	7424	$5D	93	23808	$9D	157	40192	$DD	221	56576
$1E	30	7680	$5E	94	24064	$9E	158	40448	$DE	222	56832
$1F	31	7936	$5F	95	24320	$9F	159	40704	$DF	223	57088
$20	32	8192	$60	96	24576	$A0	160	40960	$E0	224	57344
$21	33	8448	$61	97	24832	$A1	161	41216	$E1	225	57600
$22	34	8704	$62	98	25088	$A2	162	41472	$E2	226	57856
$23	35	8960	$63	99	25344	$A3	163	41728	$E3	227	58112
$24	36	9216	$64	100	25600	$A4	164	41984	$E4	228	58368
$25	37	9472	$65	101	25856	$A5	165	42240	$E5	229	58624
$26	38	9728	$66	102	26112	$A6	166	42496	$E6	230	58880
$27	39	9984	$67	103	26368	$A7	167	42752	$E7	231	59136
$28	40	10240	$68	104	26624	$A8	168	43008	$E8	232	59392
$29	41	10496	$69	105	26880	$A9	169	43264	$E9	233	59648
$2A	42	10752	$6A	106	27136	$AA	170	43520	$EA	234	59904
$2B	43	11008	$6B	107	27392	$AB	171	43776	$EB	235	60160
$2C	44	11264	$6C	108	27648	$AC	172	44032	$EC	236	60416
$2D	45	11520	$6D	109	27904	$AD	173	44288	$ED	237	60672
$2E	46	11776	$6E	110	28160	$AE	174	44544	$EE	238	60928
$2F	47	12032	$6F	111	28416	$AF	175	44800	$EF	239	61184
$30	48	12288	$70	112	28672	$B0	176	45056	$F0	240	61440
$31	49	12544	$71	113	28928	$B1	177	45312	$F1	241	61696
$32	50	12800	$72	114	29184	$B2	178	45568	$F2	242	61952
$33	51	13056	$73	115	29440	$B3	179	45824	$F3	243	62208
$34	52	13312	$74	116	29696	$B4	180	46080	$F4	244	62464
$35	53	13568	$75	117	29952	$B5	181	46336	$F5	245	62720
$36	54	13824	$76	118	30208	$B6	182	46592	$F6	246	62976
$37	55	14080	$77	119	30464	$B7	183	46848	$F7	247	63232
$38	56	14336	$78	120	30720	$B8	184	47104	$F8	248	63488
$39	57	14592	$79	121	30976	$B9	185	47360	$F9	249	63744

Hex	LSB	MSB	Hex	LSB	MSB	Hex	LSB	MSB	Hex	LSB	MSB
$3A	58	14848	$7A	122	31232	$BA	186	47616	$FA	250	64000
$3B	59	15104	$7B	123	31488	$BB	187	47872	$FB	251	64256
$3C	60	15360	$7C	124	31744	$BC	188	48128	$FC	252	64512
$3D	61	15616	$7D	125	32000	$BD	189	48384	$FD	253	64768
$3E	62	15872	$7E	126	32256	$BE	190	48640	$FE	254	65024
$3F	63	16128	$7F	127	32512	$BF	191	48896	$FF	255	65280

Appendix D
6510 OP codes

The tables presented in this appendix give the 6510 Op Codes, with the hex codes for each mnemonic categorised by address mode.

The notation is as follows:

A	Accumulator
X	Register X
Y	Register Y
S	Stack Register
PC	Program Counter
P	Processor Register (Status Flags)
M	Memory Location

The P Register flags are:

Bit 0	**C**	Carry
1	**Z**	Zero
2	**I**	Interrupt
3	**D**	Decimal
4	**B**	Break
5	*	(not used)
6	**V**	oVerflow
7	**N**	Negative

6510 Op Codes

MNEMONIC	DESCRIPTION	IMPLIED	ACCUM	IMMED	ZERO PAGE	ABSOLUTE ADDRESS
ADC	ADD WITH CARRY			69	65	6D
AND	LOGICAL AND			29	25	2D
ASL	ARITHMETIC SHIFT LEFT		0A		06	0E
BCC	BRANCH ON CARRY CLEAR					
BCS	BRANCH ON CARRY SET					
BEQ	BRANCH IF EQUAL TO ZERO					
BIT	COMPARE BITS WITH ACCUMULATOR				24	2C
BMI	BRANCH ON MINUS					
BNE	BRANCH ON NOT EQUAL TO ZERO					
BPL	BRANCH ON PLUS					
BRK	BREAK	00				
BVC	BRANCH ON OVERFLOW CLEAR					
BVS	BRANCH ON OVERFLOW SET					
CLC	CLEAR CARRY	18				
CLD	CLEAR DECIMAL	D8				
CLI	CLEAR INTERRUPT MASK	58				
CLV	CLEAR OVERFLOW FLAG	B8				
CMP	COMPARE TO ACCUMULATOR			C9	C5	CD
CPX	COMPARE TO REG-X			E0	E4	EC
CPY	COMPARE TO REG-Y			C0	C4	CC
DEC	DECREMENT MEMORY				C6	CE
DEX	DECREMENT REG-X	CA				
DEY	DECREMENT REG-Y	88				
EOR	EXCLUSIVE OR ACCUMULATOR			49	45	4D
INC	INCREMENT MEMORY				E6	EE
INX	INCREMENT REG-X	E8				
INY	INCREMENT REG-Y	C8				
JMP	JUMP TO ADDRESS					4C

ABSOLUTE X	ABSOLUTE Y	Z-PAGE X	Z-PAGE Y	RELATIVE ADDRESS	INDIRECT	INDIRECT X	INDIRECT Y	P-REGISTER N	V	*	B	D	I	Z	C
7D	79	75				61	71	×	×					×	×
3D	39	35				21	31	×						×	
1E		16						×						×	×
				90											
				B0											
				F0											
								M7	M6					×	
				30											
				D0											
				10											
											×				
				50											
				70											
												0			0
													0		
									0						
DD	D9	D5				C1	D1	×						×	×
								×						×	×
								×						×	×
DE		D6						×						×	
								×						×	
								×						×	
5D	59	55				41	51	×						×	
FE		F6						×						×	
								×						×	
								×						×	
					6C										

6510 Op Codes

MNEMONIC	DESCRIPTION	IMPLIED	ACCUM	IMMED	ZERO PAGE	ABSOLUTE ADDRESS
JSR	JUMP TO SUBROUTINE					20
LDA	LOAD ACCUMULATOR			A9	A5	AD
LDX	LOAD REG-X			A2	A6	AE
LDY	LOAD REG-Y			A0	A4	AC
LSR	LOGICAL SHIFT RIGHT		4A		46	4E
NOP	NO OPERATION	EA				
ORA	INCLUSIVE OR ACCUMULATOR			09	05	0D
PHA	PUSH A ONTO STACK	48				
PHP	PUSH P-REGISTER ONTO STACK	08				
PLA	PULL ACCUMULATOR FROM STACK	68				
PLP	PULL P-REGISTER FROM STACK	28				
ROL	ROTATE LEFT ONE BIT		2A		26	2E
ROR	ROTATE RIGHT ONE BIT		6A		66	6E
RTI	RETURN FROM INTERRUPT	40				
RTS	RETURN FROM SUBROUTINE	60				
SBC	SUBTRACT WITH CARRY			E9	E5	ED
SEC	SET CARRY	38				
SED	SET DECIMAL	F8				
SEI	SET INTERRUPT MASK (DISABLE)	78				
STA	STORE ACCUMULATOR				85	8D
STX	STORE REG-X				86	8E
STY	STORE REG-Y				84	8C
TAX	TRANSFER A TO X	AA				
TAY	TRANSFER A TO Y	A8				
TSX	TRANSFER S TO X	BA				
TXA	TRANSFER X TO A	8A				
TXS	TRANSFER X TO S	9A				
TYA	TRANSFER Y TO A	98				

ABSOLUTE X	ABSOLUTE Y	Z-PAGE X	Z-PAGE Y	RELATIVE ADDRESS	INDIRECT	INDIRECT X	INDIRECT Y	P-REGISTER N	V	*	B	D	I	Z	C
BD	B9	B5				A1	B1	×						×	
	BE		B6					×						×	
BC		B4						×						×	
5E		56						0						×	×
1D	19	15				01	11	×						×	
								×						×	
								×	×	×	×	×	×	×	×
3E		36						×						×	×
7E		76						×						×	×
								×	×	×	×	×	×	×	×
FD	F9	F5				E1	F1	×	×					×	×
												1			1
													1		
9D	99	95				81	91								
			96												
		94						×						×	
								×						×	
								×						×	
								×						×	
								×						×	
								×						×	

Appendix E
Games Reviews

Before reviewing a few of the excellent games available for the '64 I would like to enter a personal plea for a new approach to computer entertainment. I would very much like to see more programmers getting together with artists and musicians to produce more creative forms of entertainment on their computers than has been evident to date. Space invaders can no longer be called art (if it ever could). The computer should be thought of just as much as a medium for expression by artists as oils and canvas or an electric guitar. It is also interesting to see the results artists can produce within the restrictions of a particular medium, for instance character square colour graphics. This can often result in greater stimulation to creativity than the freedom available using, say, true high resolution colour graphics. Progress in this respect is perhaps more evident in France, where I have seen some of the most remarkable graphics in the computer glossies, than in the UK, and in the States people seem quite happy to pay around thirty dollars for the most mundane rubbish. It is up to programmers to a large extent to initiate education in this respect, and to design tools which the computer illiterate person can use creatively on a computer, something which very few programmers seem able, or willing, to do.

Another bugbear of mine is the preoccupation with killing, and competitiveness in general, of many computer games. There is no reason why computer games, even arcade games, should not be played with the computer, rather than against it without being any less enjoyable, exciting or satisfying.

The single exception to these criticisms I have seen to date is an American program, soon to be distributed by Audiogenic, called Alice in Videoland. (Apologies then to American programmers for the comment in the paragraph above.) It is a children's game for adults, an interactive high resolution animated cartoon version of Alice in Wonderland in which you play Alice. It is an arcade game in that it is played in real time using a joystick. The graphics are wonderful and surreal and the music seems to vary from a full fledged Bach harpsichord solo, when Alice is doing well, to a fugue or dirge when things are going badly. Even if you can't afford to buy it (I believe it comes on three discs!) do try to take a look. It is interesting to see, also, the power of the Commodore 64 being used, for the first time, to its full. I would like to see something on this scale released for other

computers on the market. There is no reason why computer entertainment should be restricted to games. Interactive, animated cartoons, as described above are not strictly games.

Attack of the Mutant Camels
Jeff Minter (Llamasoft)

Jeff Minter is, in my opinion one of the best games programmers around. "The evil alien nasties have invaded the Earth – but not in person! They have abducted some camels for Earth and used devious genetic engineering to mutate these normally harmless ships of the desert into 90 foot high, laser spitting, neutronium shielded death camels. Your mission is to pilot your tiny craft in combat against these mutants." AMC is a work of art and in a sense is a reversal of all space games that have gone before. You take one of the biggest, most harmless beasts you can think of, make it plod soulfully across a barren planet, and then pound it to death with the entire resources of modern technology.

The use of graphics and sound is superb. Jeff seems to design a new character set for each game he writes, and all the trimmings are there, high score, various options – speed of camels walking, speed and accuracy of camels firing, proportion of normal to nasty bombs, speed of rockets in hyperwarp, deresolution on collision with camels, number of players etc . . . The camels are the biggest sprites you have ever seen! The backdrop scrolls smoothly across a sky filled with glittering stars, according to your viewpoint and the camels' footsteps and the jets and lasers pound and screech and squeal eerily in the background. Jeff Minter's games are unusual in that listening to the sound is an essential way of keeping up with what is going on in the game. The camels spit two types of bolts at your ship, both of which follow you as you try to take evasive action. When you eventually learn to use the joystick well enough to be able to hold your ship stationary behind the camels, you can blast them to death, weakening their plutonium shielding (which changes colour appropriately) until they phase out of existence. In later stages of the game holding still is not enough, and you must zoom all over the screen. Attack of the Mutant Camels is not so much a game as an experience. After disposing of seven of the beasts you warp through hyperspace – this uses every trick in the book – and on to the next sector (and another seven camels). My criticisms of AMC are that there could have been more and different things at each new stage and that it is not terribly addictive purely as a game. If you allow any camel to reach the right hand side of the screen, sector defences are penetrated. Earth base is so annoyed that they push the destruct button and blow you to tiny bits. This sometimes seems to happen regardless of sector penetration.

Gridrunner and Matrix

Gridrunner (for the VIC20) was number one in the American national computer games charts for weeks. It is totally and completely addictive, you always know that you can get those extra points with just one more game, but it is very hard to explain why. It is one of the fastest games around, and you must try to grasp the entirety of what is happening on the screen. Loosely based on Centipede it seems to be an attempt to take an existing game as far as possible.

Earth has set up a huge solar power collection system known as the Grid, which has been invaded by alien Droids, who were using its power to reproduce in preparation for an invasion of the world. A special highly manoeuverable ship, the Gridrunner controlled from your joystick) was developed to combat the Droids.

The Droids have three main offensive tactics.

Gridsearch squads (centipedes) are linked Droid segments which traverse the Grid horizontally, descending whenever they reach an obstruction. If a squad is hit in the body it divides into two independent squads.

Pods are small yellow devices that lodge themselves at nodes of the Grid, eventually hurling a single lethal bolt of energy down at the end of their life cycle.

The X and Y zappers are two ships that traverse the boundaries of the Grid, periodically emitting plasma beams which destroy anything in their path, and form a new Pod wherever the two beams meet.

Your Gridrunner can move freely along the bottom seven lines of the Grid, but your ship cannot move through the pods. You will be destroyed if you are hit by a Droid, a charge from a pod or a heavy plasma beam. A sore trigger finger is guaranteed with each attempt to master the Grid. The game makes impressive use of sound and character graphics, appears to be bug free, and my copy loads every time without problems.

Matrix (or Gridrunner II) goes a stage further still. It is ten years later. The vanguished have returned in force (it's probably taken you that long to master the game). Technology has advanced on both sides. Your ship can now move all over the Grid. Scouts have reported Droids flying diagonally across the Grid, force fields that reflect your fire straight back at you, and some kind of bizarre, psychological disorientation tactics – some pilots have reported what appear to be camels running down the Matrix. . . There is also the Snitch, a traitorous humanoid who runs along the top of the Matrix and waves to the aliens to point you out!

The animation in Matrix is excellent, the screen scrolls downwards or diagonally while you are playing, creating the illusion that you are not moving in a straight line. Each of the 20 grid zones is different. Some have diagonal gridsearch squads, camels, deflectors, mystery bonuses, and some are played without a Grid at all! Any or all of these features may be

combined, and you may start from any level up to zone 6. To consider yourself good at this game you should be scoring 100,000 points and over. After weeks of practice I only score 30,000 to 50,000. You score 106 points for every cosmic cameloid you blat! Games with a sense of humour.

Laserzone
Jeff Minter (Llamasoft)

Laserzone from Llamasoft is pure shoot'em-up fun, but with a twist. You control two giant plasma cannons, one of which moves up and down the right hand edge of the screen, and the other along the bottom, controlled respectively by the forward/backward and left/right motions of your joystick. If either goes off the screen it comes back on at the other end of its track. You are being attacked by the vile Iratain skullships and the 'extremely nasty' Zzyzax bugships. If you let them reach the right hand side or bottom of the screen they will progress along the edge towards your cannon and destroy it. The right hand cannon fires across the screen and the bottom cannon fires up the screen. Most of the baddies can be caught in the cross fire, but some still sneak in from the bottom left or top right. By holding down the fire button and flicking the joystick SW the upper cannon can be made to start firing diagonally down the screen towards the bottom left, or the lower one can fire diagonally towards the top right with a NE flick. A diagonally firing cannon cannot move. Thus virtually all the invaders can be caught in the cross fire but the few which still sneak through necessitate a return to normal mode, achieved by momentarily releasing the fire button. So you must learn to control diagonal fire on demand and destroy enemies moving along the tracks towards a cannon, without shooting the cannon yourself. This takes time and a practice mode is available. Every now and then an Iratain death satellite also drifts by, releasing random bolts of energy.

You have five lives and are supplied with three 'electrobolts'. If things are getting bad, releasing an electrobolt destroys all enemies on the screen at that time. For every screen you finish you are rewarded with an extra electrobolt up to a maximum of three. As usual with Jeff's games there are options for the number of joysticks and players, 20 screens, of which you can enter any of the first nine straight away, and a new character set.

Hover Bovver
Jeff Minter (Llamasoft)

In the latest theme game from Llamasoft you don't kill anybody. Indeed the worst thing that can happen to you is to have your lawnmower taken away (and it wasn't even yours)! One reviewer commented that playing Hover

Bovver was like being involved in something out of the Keystone Kops, and indeed this is the funniest game I have ever played.

Summertime in England and Gordon Bennet's thoughts turn to mowing. Unfortunately his lawnmower has rusted beyond repair so he borrows his neighbour's. Without asking of course. Jim decides that he wants his AirMo back and sets out in hot pursuit, while Gordon, controlled by your joystick, must mow as many of his (many) lawns as he can without being caught. If he is caught he loses the mower and has to resort to stealing from one of his two other neighbours.

You (Gordon), Rover (Gordon's dog), the neighbours or the gardener cannot pass through the hedges in the gardens. Rover normally wanders around doing doggy things but is annoyed by the noise of the mower. When he cannot stand it any longer he will give chase and, if he catches up, attack the mower. In this case it overheats and stops until it is cool again. The faster you mow the hotter the mower gets, with the same result. If you mow over the flower beds an angry gardener will appear and give chase, taking away the mower if he catches you. Rover and the gardener would never cross a flower bed, but the neighbours are not so thoughtful. On pressing the fire button on your joystick, Rover will chase the neighbour, barking fiercely. The neighbour and the gardener will both flee from the dog if he is barking, otherwise the neighbour will keep his distance but the gardener will continue to give chase. Eventually Rover's loyalty wears thin and he will cease to obey you.

The game opens with a title page and logo. On pressing the fire button Gordon is seen sneaking surreptitiously into his neighbour's garage and emerging with Jim's AirMo. This sequence is repeated every time Gordon needs another mower and, though it shows off the multicolour layered sprite graphics of the 64 very well, it gets very tedious until you can keep possession of the AirMo long enough to need the break. The sound effects in general are quite good, especially the mower, and the whole game is played to a rousing two part piano rendition of English Country Garden. It is a pity that this can also rapidly get on your nerves, and that on my copy the option to switch the sound off does not work (you can always turn your telly down, however). All the usual Minter extras are there; one or two player, and one or two joystick options, 16 lawns of progressive difficulty, scoring on an associated scale, dog tolerance, fidelity and motor overload scales at the bottom of the screen, entry at any lawn from 1 to 8, freeze frame and highscore.

The view of the action is from above and the animation is wonderful. The dog and the pursuers appear to have an independent intelligence of their own and will often do the most incredibly lifelike and funny things. Rover wanders around sniffing things, gets bored and pootles off, ignoring the neighbour who nevertheless gets paranoid and will do anything to avoid contact with the dog. I also discovered that it is possible to mow a maze in

the flower beds into which you can get the gardener temporarily trapped. The action is quite hectic, but this is essentially a game of strategy and you often get the chance to retire to one end of the garden for a good ponder. For instance, a stable situation will occur where Rover is trying to get at the lawnmower but will not cross the flower beds, and the neighbour will not approach any closer because the dog is in between. Though sometimes you may get an apparently stable situation with the neighbour waving his arms at you over a hedge, when Rover will suddenly decide to go round and chase the neighbour out, and you end up with them both after you! Recommended without reservations.

Appendix F
Glossary of Computer Terms

PIXEL: The smallest point which can be plotted on a computer screen.

BIT MAPPED GRAPHICS: A graphics mode in which each pixel on the screen corresponds to (at least) one bit in graphics memory.

HIGH RESOLUTION GRAPHICS: A graphics mode with a resolution of at least 256 by 192 Pixels, usually bit mapped. The Commodore 64 uses 320 by 200 pixels with up to 16 colours on the screen simultaneously.

TRUE HIGH RESOLUTION COLOUR GRAPHICS: A bit mapped high resolution colour graphics mode in which each pixel on the screen can be independently set to any colour in a specified pallette (set of at least four colours). Very few home computers have such a mode, as it is very expensive on memory. At the time of writing the only machines I know of, with a mode satisfying these requirements, are the BBC computer and the Lynx. For instance a 256 by 192 pixel screen with 16 colours requires:

$(256 \times 192 \times 4)/(8 \times 1024) = 24$k bytes RAM

Surprisingly, colour is a lot less expensive on memory than resolution. Doubling the horizontal resolution of the screen doubles the amount of memory required to store it, whereas changing the number of colours from 16 to 32 would only involve a 25% increase in memory.

CHARACTER SQUARE HIGH RESOLUTION COLOUR GRAPHICS: A bit mapped high resolution colour graphics mode in which each (usually 8 by 8 pixel) character square can contain a small number of colours (usually two) from a specified pallette. This is probably the most useful compromise between colour resolution and memory and is used by the Commodore 64 and the Sinclair Spectrum. Machines such as the Memotech and the Oric Atmos use similar fiddles.

CHARACTER GRAPHICS: A non bit-mapped graphics mode in which screen memory contains an array of around 40 by 25 pointers to a set of shapes which are displayed within the corresponding character squares on the screen. Efficient on memory and useful for animation. Indeed on

machines such as the Sinclair Spectrum and the Oric Atmos, it is the main means of animation.

ROM: Read Only Memory. On the Commodore 64 this contains the BASIC interpreter, the character set and the operating system. Thus these cannot be changed without first copying them into RAM.

RAM: Read and Write memory. The initials actually stand for Random Access Memory, but as both ROM and disk memory may be randomly accessed the name is perhaps slightly misleading.

SERIAL FILE: A tape or disk file which can only be read from beginning to end.

RANDOM ACCESS FILE: A disk file which may be read in any order. Commodore call them relative files.

BIT: The smallest unit of memory (ROM or RAM) in a computer.

BYTE: The smallest addressable unit of memory in a computer. Usually eight bits on a home micro, it can contain any number between 0 and 255.

NYBBLE: Half a byte (four bits), which is enough memory to store colour information on the Commodore 64.

INTERPRETER: Commodore 64 BASIC is interpretive. The ROM contains a machine language program which treats your BASIC program as a set of instructions and obeys them. This is slow and inefficient as the microprocessor spends most of its time obeying the interpreter, and hardly any obeying your program.

COMPILERS: Source languages such as FORTRAN, PASCAL, ALGOL 60 and ALGOL 68 are normally compiled, that is translated into machine language, before they are run. The compiled, or object, programs run quickly and efficiently, but this is a very slow method of developing a program as each new version must be compiled before it can be tested. Compilers are available for BASIC source code. Ideally program development should be carried out using an interpreter, and then the finished program compiled for efficiency. Compilers also tend to produce rather large object code which can be a problem on a small micro.

THREADED LANGUAGES: FORTH and LISP use a remarkable compromise between the two methods above. The source code is translated line by line at entry time into a very efficient intermediate code (in a similar way

to that in which BASIC is actually translated into tokens which is then recursively interpreted by a very small and fast interpreter. Commodore's pseudocompiler PETSPEED uses a similar technique. Unfortunately, as FORTH is stack based (hence its speed) the input is in the rather unfamiliar Reverse Polish notation, though there is no reason why a preprocessor could not be written to handle normal syntax entry.

STRUCTURED LANGUAGES: FORTRAN and some types of BASIC take only one command per line and make heavy use of jumps and conditional (IF) jumps to control program flow. BASIC is the only language which does not allow local parameter passing between procedures but is restricted to the use of global variables. Structured, procedural languages such as PASCAL, ALGOL, CORAL and structured BASICs, such as BBC BASIC, allow a much more English-like modular syntax with nested conditional statements (IF . . . THEN . . . ELSE IF . . .) and nested user defined procedures. This is generally acknowledged to lead to good programming practice.

The Complete Reference Companion for all Commodore 64 Users

This **Companion** details the complex capabilities of the Commodore 64 as it extensively develops and expands the coverage in the User Manual. Detailed analysis of the micros graphics and music capabilities reveal uncharted gaps in the graphics memory map and unearths the orchestra you never suspected lay buried in the sound chip.

The music synthesiser program together with the graphics programs allowing you to create and store bit-mapped graphics in both hi-res and multi-colour modes, help lay bare the hidden potential of the 64. With the aid of these large programs, each utilising invaluable subroutines transferable into your own programs, the facilities of the 64 are investigated bit by bit and byte by byte.

Routines for interrupt-driven sound lead on to the use of raster interrupts for animation and the spawning of up to 72 sprites, with all the routines given in both BASIC and assembler forms.

A boon to anyone interested in getting to grips with their machine, this book will give the user the tools needed to expand the horizons of their 64.

0 330 28479 7

U.K. **£5.95**