machine code graphics and sound

for the commodore 64

easy to load routines and ideas

mark england and david lawrence



machine code graphics and sound

for the commodore 64

easy to load routines and ideas

mark england and david lawrence

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

Copyright © Mark England & David Lawrence, 1984

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

British Library Cataloguing in Publication Data
Mark England, 19---
Machine code graphics and sound for the Commodore 64.
1. Commodore 64 (Computer)——Programming
2. Computer sound processing
3.Computer graphics 4.Machine codes (Electronic computers)
I. Title II. Lawrence, David
001.64'2 QA76.8.C64

ISBN 0–946408–28–9

Cover design by Grad Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

	Page
Foreword	vii
How to use this book	1
1 Utility Routines	3
2 Low Resolution Commands	11
3 High Resolution Commands	49
4 Sprite Commands	125
5 Sound Commands	161
6 The BASIC Extender	201
APPENDIX A: Hex Loader	211
APPENDIX B: Merge Program and Control Characters	216

Contents in detail

CHAPTER 1

Utility Routines

Essential routines which must be resident in memory before new commands are added: GETWRD, GETBYT, GETSTRING, Error routine additions.

CHAPTER 2

Low Resolution Commands

Understanding low resolution, the position of the screen in memory, character data — BANK, changing the video bank — LOCATE, positioning the cursor — SCREEN, the position of screen memory — COLOUR, setting the screen colour — PLOT, plotting a low resolution pixel — ALLOT, reserving memory — CCOPY, copying the character set — CHRPTR, moving the character data pointer.

CHAPTER 3

High Resolution Commands

Understanding high resolution, organisation of the high resolution screen — LOW, TURNOF, HIGH, setting up the high resolution screen — HSCREEN, setting up screen parameters — HCLEAR, clearing the high resolution screen — HLOCATE, locating the high resolution cursor — HPLOT, plotting pixels in high resolution — INTEGER, WINDOW, LINE PLOTTING, high resolution line drawing — CIRCLE, high resolution circles — TEXT, PUTCHR, high resolution text.

CHAPTER 4

Sprite Commands

Sprites explained — sprite parameters — SPRITE, switching on a particular sprite — DESPRITE, switching off a sprite — SCOLOUR, setting a sprite to any colour — ENLARGE, enlarging a sprite — SPTR, changing a sprite — RESPRITE, defining a sprite simply — SHAPE, defining a sprite from within a BASIC program.

CHAPTER 5

Sound Commands

Sound in general — BEEP, specifying a single note of specified duration — CHORD, three simultaneous beeps — advanced sound commands — GETPAR, picking up sound parameters from the BASIC text — VOLUME, affecting output of all three channels — ADSR, attack, decay, sustain and release set in one statement — FILTERS, setting up type of filter, filter resonance factor and filter breakthrough — VOICE, voice control — PULSE, widening the pulse width — PLAY, composing and playing music on the 64.

CHAPTER 6

The BASIC Extender

How to make the previous routines fully part of the BASIC language, each with its own keyword.

Foreword

Those who have already bought and used our previous book, *Commodore 64 Machine Code Master* (Sunshine Books, 1983) will need little introduction to what this book contains. In *Code Master* we showed how the immense flexibility of the Commodore 64 in all things machine code could be employed to extend the boundaries of the normal BASIC language. We chose to extend BASIC because we felt that there were too many books on the micro market which provided machine code routines and advice which no-one would ever actually use. Our intention was to provide routines which illustrated machine code methods and yet which could be used in day-to-day programming by those who had only just begun the long trek away from BASIC.

When we had finished *Code Master*, we had the feeling that the job was not finished. Reactions to that book, both from readers and reviewers, showed that the approach and style were right but we were aware that so much still remained to be done. The greatest strengths of the 64 are in the fields of sound and graphics, and these we had hardly touched upon at all. Out of that dissatisfaction, the present book was born.

Like *Code Master*, this is not a book of theory, nor is it a beginner's introduction to machine code — it is a book which shows how machine code can be set to work. Unlike *Code Master*, which gave over half the book to the listing of the Mastercode Assembler, this book is *all* practical routines to alter the face of the machine. Anyone who works their way systematically through the book will find that they have released sound and graphics capabilities that previously were the reserve of the skilled games program writers. In addition, as the methods we use become familiar, readers (or rather *users*) will find that they have begun to grasp how practical machine code is written and used.

But perhaps what readers will learn most of all, as we have in writing, is just what an extraordinary machine the Commodore 64 really is. Many have pointed to its BASIC dialect, which is admittedly beginning to show some signs of creaking with age. Few seem to have seen the other side of the coin — that here is a machine which, surely more than any other on the market, is designed for those who want to go beyond BASIC. The 64 often seems more like a chameleon than a moderately-priced personal micro, for there is almost nothing about it apart from the colour of the box that the user cannot change. It is the flexibility of the 64 which has made this book possible, and its outstanding sound and graphics capabilities that have made it worthwhile.

INTRODUCTION
How to use this book

The overall object of this book is to provide you with a series of machine code routines, in the fields of sound and graphics, which can eventually be added to the BASIC language in the form of new keywords. In order to achieve this, it is important that you use the book correctly.

1. Entering the routines

Before embarking on the main part of the book you will need one of two things, either an assembler program of some kind or a more simple tool to allow you to enter machine code directly into memory. Of the two, an assembler will make the task considerably easier. The routines in this book were all written using the Mastercode assembler which began its life in our first book and is now available commercially from Sunshine Books. Readers of the first book will be able to use the version of Mastercode which was given there.

If you do not have and are not prepared to get an assembler, then you will have to fall back on entering machine code directly in some form or other. At the back of this book, in Appendix A, you will find that there is a very friendly little Hex Loader which will not only allow you to enter the machine code from tables given in the book but will also check your entries with a high degree of accuracy. Before using the Hex Loader, do take the trouble to read the whole of the appendix — you may save yourself some work later on.

2. Parameters

Because we shall be creating genuine BASIC commands, which like normal BASIC will allow values to be specified alcngside them in a BASIC line, the first step in using any of the routines in this book is to enter the parameter routines from Chapter 1.

3. Extent of code in memory

There is an awful lot of code in this book — something more than 4K for the command routines themselves. There is nothing wrong in principle with trying to enter all the routines in a single chunk but the inevitable

mistakes you will make, with the consequent need for reloading the assembler/Hex Loader and the code itself, can make this frustrating. The method we recommend is that you begin each of the four chapters which contain the actual commands with only the parameter routines in memory (or as part of the Hex Loader — see Appendix A).

4. Order of entry

Work through the commands in each section *in order*. While it is not always the case that routines depend for their working on others which precede them in the same section, it happens often enough to make it inadvisable to depart from the order we give.

This does not apply to routines from *other* sections. Apart from the fact that all the commands use the parameter routines in Chapter 1, there is no interaction between the commands in different sections at all. Thus, provided that the parameter routines are entered first for each, you can have a separate set of low resolution and high resolution sprite and sound commands.

5. Converting the BASIC

As you enter the separate commands you will find that they can only be used by means of SYS commands, ie they can only be run like a machine code routine, albeit a machine code routine that works just like a BASIC command. Have patience. Once you have entered all the routines, testing as you go, then is the time to move on to the BASIC Extender which will integrate the commands into normal BASIC. Up until that stage, there is no point in trying to use any of the new keywords that you will find in the book, they will simply result in a syntax error. You do not, however, *have* to enter and use the new keywords. Provided that the routines have been successfully tested, you can use them, in exactly the same way by employing the SYS command.

6. The final stages

When you have four separate files holding the commands from the four major sections of the book, you may want to think about combining your work into one version of the Hex Loader, or one machine code file for ease of loading and saving. Suggestions for this are given in Chapter 6.

CHAPTER 1
Utility Routines

In this section, we examine a number of uncomplicated routines which must be resident in memory before any new commands are added.

GetWrd

One of the main purposes of this book, like its predecessor, *Commodore 64 Machine Code Master*, is to provide the reader with examples of practical machine code routines which can make a difference to everyday programming. With that in mind, although you may never notice them at work, the first three routines are a vital link in the chain. Almost all of the new commands you will add to your BASIC in the course of this book need parameters to work upon. That is to say, they will not only tell the 64 to perform an action but they will also provide some extra information about the *way* in which the action is to be performed, for instance GOTO 100, where the '100' is a vital part of the command.

Rather than adopt the clumsy method of POKEing the necessary values into unused parts of memory where the new machine code routines can find them, we have chosen to make all the routines work in the same way as normal BASIC by picking up the information they need from a line in the BASIC program. In addition, mimicking BASIC in this way will allow the user to employ expressions as parameters, eg PRINT A + B + C, a practice which is essential to effective programming. In actual fact this is not as difficult as it sounds as the 64's BASIC interpreter, the massive machine code program which creates the BASIC language, already contains the necessary routines.

The first routine, GETWRD (abbreviated from GET WORD), is designed to pick up from a BASIC line a value in the range 0–65535 or, in computing terms, a 16-bit 'word'. In fact, the routine consists of nothing more than two calls to routines in the ROM. All the work of calculating where in the BASIC line the parameter will be found and what to do if it is not there is undertaken by the BASIC interpreter. Whether we have called up a command using SYS or completely integrated it with BASIC by giving it its own keyword (see 'How to use this book'), the parameter will be taken from the place in the BASIC line immediately following the call, ignoring spaces, of course. Having been picked up from the line, the resulting value

is placed into a two-byte register at 20–21 ($14–15) in zero-page memory. You will constantly come across references to $14–15 in the sections that follow and, when you do, you will know that what is being dealt with is almost always a numerical parameter picked up from a BASIC line.

GetWrd — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 SYM
00                 30 ORG $C000
C000               50 GETWRD
C000   208AAD       60 JSR $AD8A
C003   4CF7B7       70 JMP $B7F7
C006               80 END

 TOTAL ERRORS IN FILE ---- 0


GETWRD            C000
 TOTAL NUMBER OF SYMBOLS ---- 1
```

Machine Code

```
ADD    DATA                          CHECKSUM

C000   20 8A AD 4C F7 B7              15DD
```

Commentary

Line 60: This is the call to the ROM routine which picks up a floating point value from a BASIC line.

Line 70: Having picked up the value, this call is to another routine which examines the parameter obtained, to see that is in the range 0–65535. If it *is* in the correct range, then the value is converted from floating point representation (which takes up five bytes of memory) to integer, which requires only two for a number in this range. If not, an ILLEGAL QUANTITY error message is automatically generated by the 64.

Testing

GETWRD is never normally called from BASIC directly, but, for the purposes of testing, enter the following lines of BASIC:

```
10 SYS(49152) 50000
20 SYS(49152) 100000
```

RUN the program and you should find that you receive an ILLEGAL

RUN the program and you should find that you receive an ILLEGAL QUANTITY error message for line 20. Line 10 is accepted because the parameter is within the acceptable range of 0–65535.

GetByt

Just as some commands work with numbers in the range 0–65535, others are limited to 'single byte' values, that is numbers in the range 0–255 which would fit into a single memory byte.

GETBYT uses the previous routine, GETWRD, to pick up a value from the BASIC line, but then checks to see that the resulting value is in the range 0–255. If not, once again an ILLEGAL QUANTITY error message is generated.

GetByt — assembly language listing

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $C006
C006              30 SYM
C006              70 GETWRD = $C000
C006              90 GETBYT
C006  2000C0     100 JSR GETWRD
C009  A515       110 LDA $15
C00B  D003       120 BNE IOERR
C00D  A514       130 LDA $14
C00F  60         140 RTS
C010             160 IOERR
C010  4C48B2     170 JMP $B248
C013             180 END

 TOTAL ERRORS IN FILE --- 0


GETWRD           C000
GETBYT           C006
IOERR            C010
 TOTAL NUMBER OF SYMBOLS --- 3
```

Machine Code

```
ADD    DATA                          CHECKSUM

C006   20 00 C0 A5 15 D0 03 A5        36E3
C00E   14 60 4C 48 B2                 06B2
```

Commentary

Lines 110–120: As mentioned earlier, the value picked up by GETWRD is

left in the memory at $14–15. If the value is in the range 0–255, then the second of these bytes *must* be zero since the number would be stored with its most significant byte second. If it is not zero, then a jump is made to the end of the program where the ILLEGAL QUANTITY error message will be called.

Line 130: On return from GETBYT, the single-byte value from the BASIC line is also stored in the accumulator.

Line 170: Error messages are printed by a routine in the ROM. According to where the routine entered, a different error message will be printed. This particular entry point will print the ILLEGAL QUANTITY error message, stop the program and return to BASIC.

Testing

As with GETWRD, this routine will not normally be called from BASIC, but the following lines will suffice to give a rough and ready test of it:

```
10 SYS(49158) 100
20 SYS(49158) 1000
```

Once again, the first line should run without problem and the second should stop with an ILLEGAL QUANTITY error message.

GetString

We can now deal with all the kinds of quantities which are likely to be required as parameters for our new commands — or at least parameters within the range of normal BASIC. There remains the problem of another kind of parameter, a string, as in MID$(A$,3), where the system must not only be capable of detecting and using the '3', but also of realising what is meant by the reference to A$. The current routine, GETSTRING, is designed to give our new commands that ability. Once again, this is a very simple routine because we make full use of the facilities of the 64's ROM.

GetString — assembly language listing

```
ADD.   DATA     SOURCE CODE
00                10 PRT
00                20 SYM
00                30 ORG $C089
C089   A9FF       90 GETSTR LDA #$FF
C08B   850D      100 STA $D
C08D   850E      110 STA $E
```

```
C08F   A96A       150 LDA #$6A
C091   A200       160 LDX #$00
C093   8549       170 STA $49
C095   864A       180 STX $4A
C097   4CB1A9     200 JMP $A9B1
C09A               210 END

 TOTAL ERRORS IN FILE ---- 0


GETSTR            C089
 TOTAL NUMBER OF SYMBOLS --- 1
```

Machine Code

```
ADD    DATA                          CHECKSUM

C089   A9 FF 85 0D 85 0E A9 6A       ABCC
C091   A2 00 85 49 86 4A 4C B1       6CD1
C099   A9                           00A9
```

Commentary

Lines 90–110: The function of locations $D and $E in the 64's zero-page memory is to indicate to the 'expression evaluator' in ROM what kind of expression is under consideration. The value 255 ($FF) indicates that what is about to be dealt with is a string.

Lines 150–180: Locations $49 and $4A are a pointer to the location in memory of the key to the current variable. In this case, we have decided to record the length and name of the string variable being picked up in the three spare bytes $6A–6C, so the address $6A is loaded into the pointer.

Line 200: It is this call to the BASIC ROM which makes the GETSTRING routine so short. The jump is to part of the LET routine in BASIC. When a string is evaluated in an expression such as:

A$ = "ABCDEF"

or

LET A$ = B$

the LET routine clearly needs some means of determining the value of whatever is on the righthand side of the equation. It does this by calling the expression evaluator routine in the ROM, finding any necessary variables in memory and, if necessary, flagging any errors in the syntax. At the end of the process, $49 and $4A will contain a two-byte address which is a

pointer to the position in memory of yet another pointer, this time in the variables area. This second pointer, which consists of three bytes, records the length of the resultant string and its actual starting address in yet another area of memory — the string memory.

All that is done here is to jump into the LET routine at the point where it begins the process of evaluation, and to allow it to do the work of determining the value of the string parameter and placing the resultant length and address in the string memory into the bytes $6A–6C indicated by the pointers $49–4A. As a result, LET has provided us with a full evaluation of the string parameter, recorded its length for us and identified the position at which it has been stored in the string memory. All the important information has therefore been recorded so that it can be used by the routine which calls GETSTRING.

Testing

Like the previous two routines, GETSTRING can be roughly checked by seeing whether it will accept a valid string expression and reject an invalid one. Enter these three lines of BASIC:

```
10 A$="ABCDEFGHIJ"
20 SYS(49289) MID$(A$,3,5)
30 SYS(49289) Q
```

No problem should be encountered with line 20, but line 30 should produce a TYPE MISMATCH error message, since the routine requires a string and encounters a numeric variable.

Error routine additions

Because we are going to extend the command set of the 64, we shall run up against circumstances where errors will be generated which will not be recognised by the error routine tailored to the existing BASIC. The object of the current routine is to add two messages to the current set, along with the criteria which will call them up. You are not expected at this stage to understand the error messages — that understanding will only come as you enter the new commands that require them.

Errors — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C09A
C09A               30 SYM
C09A               70 ERRADD = $0022
C09A              100 BANKER
```

```
C09A  A9AC       110 LDA #MSG1-MSG1/256*2
56
C09C  A2C0       120 LDX #MSG1/256
C09E  4CA5C0     130 JMP ERROR
C0A1             150 SERROR
C0A1  A9B6       160 LDA #MSG2-MSG2/256*2
56
C0A3  A2C0       170 LDX #MSG2/256
C0A5             190 ERROR
C0A5  8522       200 STA ERRADD
C0A7  8623       210 STX ERRADD+1
C0A9  4C47A4     220 JMP $A447
C0AC             240 MSG1
C0AC  57524F     250 BYT 87.82.79.78.71.3
2.66.65.78.75+128
C0B6             260 MSG2
C0B6  4F5554     270 BYT 79.85.84.32.79.7
0.32.83.80.82
C0C0  495445     280 BYT 73.84.69.32.68.6
5.84.65+128
C0C8             290 END

 TOTAL ERRORS IN FILE --- 0


ERRADD           22
BANKER           C09A
SERROR           C0A1
ERROR            C0A5
MSG1             C0AC
MSG2             C0B6
 TOTAL NUMBER OF SYMBOLS --- 6
```

Machine Code

```
ADD   DATA                        CHECKSUM

C09A  A9 AC A2 C0 4C A5 C0 A9     A6DD
C0A2  B6 A2 C0 85 22 86 23 4C     A78A
C0AA  47 A4 57 52 4F 4E 47 20     60DE
C0B2  42 41 4E CB 4F 55 54 20     4C44
C0BA  4F 46 20 53 50 52 49 54     46DE
C0C2  45 20 44 41 54 C1           0F2D
```

Commentary

Lines 110–130: These instructions will be called by the later routines relating to the bank currently addressed by the VIC chip. Their effect will be to place the address of a bank error message into registers A and X, then jump to the main part of the additional error routine.

Lines 160–170: The same function as the two previous lines, but for an error relating to the sprite handling routines.

Lines 190–220: The address of the relevant error message is placed into $22–3 (ERRADD), which is used by the BASIC error routine, and then that routine is called. The BASIC error routine automatically tidies up temporary storage areas in memory, clears the return stack and prevents the entering of CONTINUE.

Lines 240–270: These lines of single-byte values represent the ASCII values of the individual messages. The function of the BYT directive is to place the values specified directly into memory without alteration of any kind. Note that the value of the last character of each message has 128 added to it. This is the signal to the error routine in BASIC that the end of the error message has been reached.

Testing

Enter SYS 49306 in direct mode, and you should see ?WRONG BANK ERROR displayed on the screen. SYS 49313 should display an OUT OF SPRITE DATA error message.

Note on the further use of this routine

The method used here can be used generally to add new error messages to BASIC for your own machine code routines. Having defined your error messages in memory, simply call the error routine starting at $C0A5, with the address of the relevant message contained in registers A and X.

CHAPTER 2
Low Resolution Commands

1. UNDERSTANDING LOW RESOLUTION

In this chapter, before moving on to look at specific new commands to control the low resolution screen, we shall examine some of the ins and outs of low resolution, especially the relationship between the low resolution screen and the rest of the 64's operating system, including the VIC chip.

The 64's low resolution display, the screen which greets the user on power-up, is a strange mixture of fine detail and crude blocks. To understand what this means, take a look at a single letter of text on the screen. You will find that it is made up of a large number of small dots, even though the quality of the television you are using may make them appear slightly fuzzy. The presence of these dots is proof that your 64 is capable of handling quite tiny units on the screen. Each of the letters and symbols in the 64's character set is made of a grid of 8*8 spaces on the screen, each space just big enough for one pixel. The only difference between the various characters is which of the pixels is turned on in the 8*8 space when the particular character is printed.

Along with this degree of detail goes the limitation that, in low resolution mode, the pixels which appear in one of the 1000 character positions on the screen are entirely beyond the control of the user. Characters may be swapped for other characters in the existing character sets or it is even possible for the user to define new character sets. The contents of any square on the screen however, can *only* be one of the shapes defined in one of the character sets, whether a predefined set or one created by the user.

What is actually seen in an individual character square is the result of the interaction between the VIC chip, an area of memory known as the 'character generator ROM' and the screen memory. Each of the 1000 bytes of the low resolution screen memory (which normally sits at 1024–2023) holds a number which represents the 'screen code' of one of the characters in the character set currently being used. It is the task of the VIC chip to scan the screen memory and then to extract from the character generator ROM the details of the shape of the character represented by the screen code. The VIC chip takes this information and uses it in creating the display which appears on the screen of the television. More will be said about this in the chapter on the high resolution screen. For the moment, it is

sufficient to hold in mind that there must be three components to provide the display:

1) An area of memory which holds the screen codes of the characters which are to appear on the screen.
2) The VIC chip, which has the job of translating the screen codes into the television display.
3) An area of memory which holds the details of the shapes of the characters.

The relationship between these three areas of memory is a very close one, not least in terms of where they must be located. This is because the VIC chip is limited to 'seeing' only one of four 16K blocks at one time. All the areas of memory with which the VIC chip has to deal — including the screen, character data and sprite data — have to be in the same 16K block, ie 0–16383, 16384–32767, 32768–49151, 49152–65535. It is this limitation of the VIC chip which leads to much of the need to rearrange memory in such a way as to fit in the data areas required.

One important component in the use of the low resolution screen remains and that is the 64's screen editing system. The 64's ROM contains a host of routines which are not so much to do with creating the display which appears on the screen, but with such matters as moving the cursor around the screen, placing new characters on to it (or removing them), taking program lines from the screen and placing them into a program, and so on. These functions are entirely separate from the work of the VIC chip and require that the 64 has a separate record of the location of the low resolution screen memory.

The position of the screen in memory

When the 64 is switched on, the screen memory is automatically located between 1024 and 2023 in memory. It does not, however, have to stay there. The screen can occupy any of the 16 1K blocks in the current video bank, provided that the VIC chip and the screen editor pointer are told where to find it. Some locations are more useful than others and some may involve a rearrangement of other competing memory uses. Placing the screen so that it begins at 2048, for instance, would have an unfortunate effect on the BASIC program, which also starts at that point — change the screen in any way and the program would be corrupted.

The location of the screen in memory is recorded for the VIC chip and the screen editor system by two different registers:

a) The VIC chip: the upper four bits of the register at 53272 tell the VIC chip in which of the 16 1K blocks it is to expect to find the screen. In the sections which follow, this register will be described by the name [SCREEN

POINTERS].

b) The Screen Editor: the register at 648 [EDITOR] stores the location of the screen for the rest of the system to consult, expressed in units of 256 bytes (ie PEEK(648)*256 represents the address at which the system thinks the screen memory is at the moment). It is quite possible that this is an incorrect value since the VIC register can be changed without affecting [EDITOR]. In this case, the screen will display data from the area of memory indicated by [SCREEN POINTERS] but there will be no cursor or possibility of manipulating the screen.

Character data

We have already noted that the VIC chip draws on an area of memory known as the character generator to get the details of the characters to be displayed on the screen. This data is permanently stored in the 64's ROM but, since the 64's ROM does not start until 40960, it could not be visible to the VIC chip's 16K-wide gaze in three of the four video banks. In fact, it is *never* visible to the VIC chip in its original position since the 64 behaves as if the section of ROM containing the character data is totally invisible and even places other data where it should be.

The VIC chip, deprived of the chance to see the character data in the ROM, makes up for it by seeing a mirage of the character data in a totally impossible position. When the 64 is first switched on, the VIC chip behaves as if it can pick up the details of the characters from an area of memory beginning at 4096. The details of the 64's two character sets require 4K of memory, so the character data apparently fills the block from 4096 to 8191. This is despite the fact that most BASIC programs will be using the memory area beginning at 2049 and a BASIC program of any size at all will use the memory well beyond 4096. This makes no difference to the VIC chip since yet another part of the system, the address decoder, when it receives a request from the VIC chip for access to any memory location between 4096 and 8191 (in video banks zero and three) actually supplies the corresponding byte in the memory area between 53248 and 57345. In other words, the VIC chip, and only the VIC chip, sees character data in this memory area, the rest of the system finds useable RAM. All of this is just another example, though one that is a little difficult to follow, of the way the 64 crams a quart into a pint pot when it comes to memory, in this case adding an effective 4K on to the 16K video banks.

Having said all this about the character data at 4096–8191, the important point to remember from the point of view of the user is that the area to which the VIC chip looks for its character data can be moved. This can be done by changing the current video bank, in which case the VIC chip will look at the same block within the new bank. If the bank is shifted to one or

three, the shadow character data are not seen by the VIC chip. Another method is to move the character data area *within* the current video bank. The character data area can be moved to begin at any 2K boundary and thus again to move away from the shadow data from the ROM. This is of particular use when redefining character sets. The position of the character data within the current video bank is controlled by the register [SCREEN POINTERS] at 53272. In this case, the lower four bits specify which of the 16 possible starting addresses is to be used.

In the sections of this chapter which follow, we shall be examining ways of manipulating all of the characteristics of the low resolution screen mentioned here, beginning with the block of memory which is the limit of the VIC chip's 16K span of attention.

2. CHANGING THE VIDEO BANK (BANK)

In this section we introduce the command BANK, which simplifies the process of moving the attention of the VIC chip from one 16K video bank to another.

Changing bank is a relatively straightforward process which involves three steps:

1) An instruction to a communication chip (CIA2) to tell it that the CPU wishes to send data to the VIC chip.
2) Sending the instruction to the VIC chip which will move the 16K video bank.
3) Informing the rest of the system, apart from the VIC chip, where the screen memory now is. Failure to do this would result in any screen editing functions, like the printing of text, being carried out on an area of memory that was no longer the screen.

Bank — BASIC listing

```
10000 REM*******************************
10001 REM CHANGE VIDEO BANK
10002 REM*******************************
10010 INPUT "BANK";VB
10020 IF VB<0 OR VB>3 THEN GOTO 10010
10025 V2=3-VB
10030 POKE 56578, PEEK (56578) OR 3
10040 POKE 56576, (PEEK(56576) AND 252)
OR V2
10050 VB=VB*64
10060 POKE 648,(PEEK(648) AND 63) OR VB
10999 END
```

Commentary

Lines 10010–10025: There are, as noted previously, four possible banks. These lines accept an input in the correct range and then invert it so that, for instance, bank zero becomes bank three. Unfortunately, the VIC chip thinks of the bank commencing at address zero in memory as being bank three and it is far more user-friendly to allow the bank number to be input in the more natural notation with bank zero at the bottom of the memory and bank three at the top.

Line 10030: The 64 contains two chips know as CIA1 and CIA2 which handle input and output between the CPU and 'external' devices such as the keyboard, the user ports, the game ports and so on. One of the jobs to which CIA2 is dedicated is communication with the VIC chip. To communicate with the outside world, the CIA chip needs first to be told whether it is being asked to output or to input, and with what device communication is to take place. With the two lowest bits of register CIA2 + 2 (56578) set, which is what the 'OR 3' does, any data coming from the CPU into the CIA2 chip will automatically flow through the two pins which are connected to these two output bits and into the VIC chip.

Line 10040: This line sets the two least significant bits of the CIA2 chip's main register at 56576 to the desired bank number, without corrupting the other bits, thus changing bank.

Lines 10050–10060: We have seen in the previous section that the 64's screen editor is entirely separate from the VIC chip and keeps its own record of where to find the screen in memory. If we are going to change the bank then we must also tell the screen editor where the screen is being moved to. In this particular case, we are only changing the bank — the position of the screen within the bank will be unchanged. What that means is that we only have to change the two most significant bits of [EDITOR] since these two bits represent units of 32K and 16K respectively.

Bank — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C0CA
C0CA               30 SYM
C0CA               70 CIA2    = $DD00
C0CA               80 GETBYT = $C006
C0CA   AD02DD     130 LDA CIA2+2
C0CD   0903       140 ORA #3
C0CF   8D02DD     150 STA CIA2+2
C0D2   2006C0     170 JSR GETBYT
C0D5   C904       180 CMP #4
```

```
C0D7   B020        190 BCS IQERR
C0D9   AD00DD      210 LDA CIA2
C0DC   29FC        220 AND #$FC
C0DE   0514        230 ORA $14
C0E0   4903        240 EOR #$3
C0E2   8D00DD      250 STA CIA2
C0E5   A514        270 LDA $14
C0E7   6A          280 ROR A
C0E8   6A          290 ROR A
C0E9   6A          300 ROR A
C0EA   29C0        310 AND #$C0
C0EC   8514        320 STA $14
C0EE   AD8802      330 LDA 648
C0F1   293F        340 AND #$3F
C0F3   0514        350 ORA $14
C0F5   8D8802      360 STA 648
C0F8   60          370 RTS
C0F9   4C48B2      380 IQERR JMP $B248
C0FC               390 END

 TOTAL ERRORS IN FILE --- 0

CIA2               DD00
GETBYT             C006
IQERR              C0F9
 TOTAL NUMBER OF SYMBOLS --- 3
```

Machine code

```
ADD    DATA                              CHECKSUM

C0CA   AD 02 DD 09 03 8D 02 DD    765D
C0D2   20 06 C0 C9 04 B0 20 AD    39DD
C0DA   00 DD 29 FC 05 14 49 03    4D2D
C0E2   8D 00 DD A5 14 6A 6A 6A    6FF6
C0EA   29 C0 85 14 AD 88 02 29    5E15
C0F2   3F 05 14 8D 88 02 60 4C    3164
C0FA   48 B2                      0142
```

Commentary

Lines 130–150: The direct equivalent of line 10030.

Lines 170–190: GETBYT is employed to pick up the bank number. The parameter is checked to see that it is in the range 0–3 and, if not, a branch is made to line 380 and from there to the routine to print ILLEGAL QUANTITY ERROR.

Lines 210–250: Equivalent to lines 10010–10025 and 10040 in the BASIC program.

Lines 270–360: These lines are equivalent to lines 10050–10060 of the BASIC program. The main difference is the method which has to be employed to translate the bank number into a start address in pages for the current bank, so that this can be placed into the upper two bits of [EDITOR]. To do this in assembly language, we rotate the value of the bank (stored in $14) right three times. In doing this the two least significant bits, the ones we are interested in, are moved first into the carry flag and then back into the accumulator at the lefthand end to become the most significant bits. The value in the accumulator can now be ANDed with $CO, or 192, since this will ensure that all but bits six and seven of the accumulator are cleared. The value is now stored once again in $14, then the current contents of address 648 are picked up in the accumulator. It is now a simple matter of clearing the upper two bits with AND, then ORing with the value in $14 and storing the result back in address 648.

Testing

BASIC: The best indication that the routine is working is that the screen is still useable when you run the routine and change the bank to two (only banks zero and two have a character set available for the VIC chip to read). If you want to make doubly sure, you could add the following temporary line:

```
10070 PRINT INT(PEEK(648)/64)
```

This should produce a value equal to the bank number you input each time the program is run. Don't forget to reset the bank to zero before you continue.

MACHINE CODE: A good test for the routine is to enter in direct mode the following line:

```
FOR I = 3 TO 0 STEP-1 : SYS (49354) I : FOR J = 0 TO 1000 : NEXT J,I
```

Note that there is *no* punctuation between the SYS address and the I in the line.

Press RETURN on this and you should see the screen instantly fill with garbage. The garbage should then change twice and, finally, the normal screen should return. Unless you are sure that something has gone seriously wrong with the routine, don't attempt to stop this procedure with the STOP key or you may have to turn your 64 off to get back to the normal screen.

Syntax and use

The correct syntax for BANK is:

```
SYS (49354) {BANK}
```

where BANK is a value in the range 0–3. *Note:* Moving the current bank to three, while perfectly possible, places the screen right over the machine code routines for extended BASIC and will probably lead to hopeless corruption.

Though changing bank is not difficult in normal BASIC, once this routine is entered into memory, the relationship between the bank number entered and the resultant addresses is much more straightforward:

BANK NUMBER	ADDRESS
0	$0000–$3FFF
1	$4000–$7FFF
2	$8000–$BFFF
3	$C000–$FFFF

When using this command it is important to remember that the VIC chip cannot see the normal character data when it is looking at banks one and three, so you must either do without them or define your own character data.

3. POSITIONING THE CURSOR (LOCATE)

Moving the cursor to a specified point on the screen is one simple capability that the 64 does not possess. The usual way around it is to use strings containing the cursor control characters. There is nothing wrong with that method: it is the one recommended in David Lawrence's *Advanced Programming Techniques for the Commodore 64.*

If we are prepared to leave pure BASIC, however, then even the non-machine code programmer can achieve the results faster by making use of a routine which is already present within the 64's ROM. If you think about it, there *must* somewhere be a section of ROM devoted to moving the cursor around, for the simple reason that the cursor can be moved. All that is not there, in the limited confines of the 8K long BASIC interpreter, is a BASIC keyword which can take the routines which are used by the cursor control characters and make use of them more directly simply to specify a screen position.

The missing keyword will now be supplied, its name being LOCATE. The procedure for LOCATE is in two stages:

1) Place the X and Y coordinates of the desired cursor position into system registers normally used for this purpose.
2) Call a routine in the 'kernal' section of the ROM which will update the cursor position based on the values contained in these registers.

Locate — BASIC listing

```
11000 REM*******************************
```

```
11001 REM PRINT AT
11002 REM*******************************
11010 INPUT "X,Y";X,Y
11015 IF X>24 OR X<0 OR Y>39 OR Y<0 THEN
11010
11020 POKE 781,X
11030 POKE 782,Y
11035 SYS (65520)
11040 PRINT "HERE"
11050 PRINT "AND THEN HERE"
11999 END
```

Commentary

Lines 11020–11030: The two registers used here are unlike almost anything else you will find in the memory. Rather than being storage locations for the 64's operating system, these two locations (and two others in the same area of memory which we do not use) are pseudo-registers representing the contents of the actual CPU registers A, X and Y. The contents of the registers are not genuinely what is in the corresponding CPU registers but, when a machine code routine is called by SYS, the values in these registers are taken into the main CPU registers before the execution of the specified machine code routine begins. Their use enables the BASIC programmer to call up all kinds of ROM routines which depend on CPU registers having certain values.

Line 11035: This SYS call is to the kernal routine called PLOT, which has two functions — to move the cursor (if the carry flag is clear) or to obtain the cursor X and Y coordinates in the CPU X and Y registers. The details do not really matter — what does matter is that, having set up the pseudo-registers with the X and Y coordinates, a call to this routine will automatically position the cursor.

Locate — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C0FC
C0FC               30 SYM
C0FC               80 GETWRD = $C000
C0FC               90 COMMA  = $AEFD
C0FC              100 GETBYT = $C006
C0FC   2006C0     130 GETPAR JSR GETBYT
C0FF   48         140 PHA
C100   20FDAE     150 JSR COMMA
C103   2006C0     160 JSR GETBYT
C106   A8         170 TAY
C107   68         180 PLA
```

```
C108  AA          190 TAX
C109  60          200 RTS
C10A  20FCC0      230 AT  JSR GETPAR
C10D  E019        240 DOAT CPX #25
C10F  B008        250 BCS IQERR
C111  C028        260 CPY #40
C113  B004        270 BCS IQERR
C115  18          280 CLC
C116  4CF0FF      290 JMP $FFF0
C119  4C48B2      300 IQERR JMP $B248
C11C              310 END

 TOTAL ERRORS IN FILE --- 0


GETWRD            C000
COMMA             AEFD
GETBYT            C006
GETPAR            C0FC
AT                C10A
DOAT              C10D
IQERR             C119
 TOTAL NUMBER OF SYMBOLS --- 7
```

Machine code

```
ADD   DATA                          CHECKSUM

C0FC  20 06 C0 48 20 FD AE 20         3470
C104  06 C0 A8 68 AA 60 20 FC         568C
C10C  C0 E0 19 B0 08 C0 28 B0         AA60
C114  04 18 4C F0 FF 4C 48 B2         2AEA
C11C                                  0000
```

Commentary

Lines 130–200: The GETBYT routine is called to pick up the vertical coordinate and the result is pushed on to the stack for temporary storage. COMMA is a ROM routine which detects whether the next character in a BASIC line is a comma — if not, a SYNTAX ERROR is flagged. GETBYT is called again to pick up the horizontal element and this is placed directly into the Y register. The vertical component is pulled back off the stack and placed in the X register.

Lines 240–270: The two values are compared with the maximum permitted values for the cursor position of 25 vertically and 40 horizontally. If either limit is exceeded, an ILLEGAL QUANTITY ERROR message is printed.

Lines 280–290: These lines call the cursor position routine in the kernal.

The carry flag is cleared before jumping to the routine because the same routine carries out two functions as mentioned in the commentary on the BASIC version. Note that, because this call is the last operation before the IQERR line, there is no need for an RTS (return from subroutine) to exit from the present routine and go back to BASIC. All that needs to be done is for execution to JMP (jump absolute) to the kernal routine, where the RTS at the end of the cursor position routine will return execution to BASIC. You will find many cases in the routines in this book where a routine apparently ends with JMP rather than an RTS to return control to BASIC.

Testing

BASIC: The BASIC routine is self-testing in that the two lines 11040–11050 print an item of text at the cursor position specified and then another at the beginning of the following line to demonstrate that the routine is working. These two lines may obviously be deleted before the routine is used in normal BASIC programming.

MACHINE CODE: Given the simplicity of the method, the only practical test is to try it out. Enter the following:

```
10 SYS (49418) 12,18
20 PRINT "XXXXX"
```

RUNning this should result in the Xs being printed in the middle of the screen.

Syntax and notes on use of LOCATE

The format for calling this command is:

SYS (49418) {ROW},{COLUMN}

where ROW is a number in the range 0–24, representing the cursor position in lines down the screen and COLUMN is a number in the range 0–39, representing character positions across the screen. SYS (49418) 12,20 moves the cursor to the middle of the screen. It is not necessary to PRINT characters immediately after calling this routine — the position of the cursor will remain where it is set until it is moved by the use of PRINT or the cursor controls.

4. THE POSITION OF SCREEN MEMORY (SCREEN)

As well as being able to move the 16K block used by the VIC chip for its overall purpose, the 64 allows the user to specify where within the current 16K block the various elements such as screen and character data fall. With this routine, we shall embark upon moving the screen within the block,

using the command SCREEN.

This command has a variety of uses — such as animation (where several slightly different screens can be used), saving the contents of a current screen or rapid changes of scene, more economical use of memory when user-defined character sets are being employed, or simply to allow a machine code program to occupy the first part of the user memory.

The procedure adopted is as follows.

1) The number of the current video bank is derived from bits six and seven of the system register at 648.
2) An error message is generated if the specified screen address is outside the current bank.
3) The relative address of the screen within the current video bank is calculated, and this is then stored in the upper four bits of the relevant VIC register.
4) This offset is recalculated into the form of 256-byte pages for storage in the system register at 648, not forgetting to preserve the value of the two highest bits, which record the current video bank.

Screen — BASIC listing

```
13000 REM*******************************
13001 REM SCREEN MOVE
13002 REM*******************************
13010 INPUT "ADDRESS";SA
13020 BANK=PEEK(648) AND 192
13030 IF INT(SA/16384)<>BANK THEN PRINT
"WRONG BANK" : GOTO 13010
13040 IF SA/1024<>INT(SA/1024) THEN PRIN
T "ILLEGAL ADDRESS" : GOTO 13010
13050 SA=SA-16384*INT(SA/16384)
13055 SA=SA/64
13060 POKE 53272, (PEEK(53272) AND 15)
OR SA
13065 SA=SA/4
13070 POKE 648,SA
13999 END
```

Commentary

Lines 13020–13030: These lines extract the current video bank from the two upper bits of the register at 648 and check that the address proposed for the screen is within that bank.

Line 13040: A check that the proposed address falls on a 1K boundary.

Lines 13050–13060: The screen address is first stripped of the value of the

video bank in which it is located and then divided by 64. Strictly, we should be dividing by 1024, which would give us the position in terms of 1K blocks. The problem is that the 0–15 value which this would give is going to be placed in the *upper* four bytes of the register at 53272 and so would have to be multiplied by 16. Far simpler to divide by 1024/16 (64) in the first place.

Lines 13065–13070: The value in SA is now the address relative to the start of the current video block divided by 64. For the system register at 648, what we need is the relative address divided by 256 (ie the address in pages). Since 256 is 64*4, all that we need to do is to divide SA again by four. The use of 'AND 192' preserves the upper two bits of 648.

Screen — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG *C147
C147               30 SYM
C147              110 GETBYT = *C006
C147              120 IQERR  = *C144
C147              130 BANKER = *C09A
C147              140 VIC    = *D000
C147   2006C0     170 SCREEN JSR GETBYT
C14A   0A         180 ASL A
C14B   0A         190 ASL A
C14C   AA         200 TAX
C14D   4D8802     210 EOR 648
C150   29C0       220 AND #*C0
C152   D021       230 BNE ERROR
C154   AD18D0     250 LDA VIC+*18
C157   290F       260 AND #*F
C159   8D18D0     270 STA VIC+*18
C15C   8A         280 TXA
C15D   0A         290 ASL A
C15E   0A         300 ASL A
C15F   0D18D0     310 ORA VIC+*18
C162   8D18D0     320 STA VIC+*18
C165   AD8802     340 LDA 648
C168   29C0       350 AND #*C0
C16A   8D8802     360 STA 648
C16D   8A         370 TXA
C16E   0D8802     380 ORA 648
C171   8D8802     390 STA 648
C174   60         400 RTS
C175              430 ERROR
C175   4C9AC0     440 JMP BANKER
C178              450 END
```

```
 TOTAL ERRORS IN FILE --- 0


GETBYT            C006
IQERR             C144
BANKER            C09A
VIC               D000
SCREEN            C147
ERROR             C175
 TOTAL NUMBER OF SYMBOLS --- 6
```

Machine Code

```
ADD   DATA                              CHECKSUM

C147  20 06 C0 0A 0A AA 4D 88            2E3A
C14F  02 29 C0 D0 21 AD 18 D0            34FC
C157  29 0F 8D 18 D0 8A 0A 0A            3426
C15F  0D 18 D0 8D 18 D0 AD 88            3532
C167  02 29 C0 8D 88 02 8A 0D            3179
C16F  88 02 8D 88 02 60 4C 9A            6162
C177  C0                                 00C0
```

Commentary

Lines 170–230: The purpose of these lines is to check that the position specified for the screen by the user is in the 16K block currently addressed by the VIC chip, by comparison with the two upper bits of [EDITOR] at 648.

Lines 250–270: The current contents of the VIC register are ANDed with 15 to wipe out the upper four bits.

Lines 280–320: The specified screen address in pages, which has been stored in the X register, is shifted left twice. This has the effect of moving the bits which record the 1K block up to occupy the four bits. This value is then placed into the VIC register by the use of OR.

Lines 340–390: Dealing with the register at 648 is actually more simple than in BASIC. The X register already contains the address in 256-byte pages within the bank; this is now placed into the lower six bits of 648.

Testing

BASIC: Run the program and move the screen to address 3072, then proceed as for the testing of the machine code routine, part b).

MACHINE CODE: a) Clear the memory of any existing program, then enter:

SYS (49479) 3

b) The screen should now be filled with garbage. Clear it with SHIFT/CLR. The computer should now work normally in respect of the usual screen functions except that, if you try to load a program longer than 1K, it will become corrupted. The reason for this is that, with the screen at the beginning at 3K and with no attempt having been made to move BASIC, the screen will be sitting right in the middle of the BASIC program, corrupting it every time a change is made on the screen. In normal circumstances, you can restore the screen to its usual position by following the procedure above and specifying 1024 as the screen address. If you do get caught with a corrupted memory, enter POKE 648,4, then press RUN/RESTORE.

Syntax and use

The syntax for the SCREEN command is:

SYS (49479) {ADDRESS}

where ADDRESS is a number between 0 and 63, representing a 1K block of memory which is within the 16K video bank currently addressed by the VIC chip. If ADDRESS is outside the current 16K video bank, a WRONG BANK error message will be generated.

5. SETTING THE SCREEN COLOUR (COLOUR)

COLOUR, the BASIC command presented here, will allow you to change screen and border colour with a simple command. There is nothing at all complex about the procedure, which is as follows:

1) Obtain the value for the screen border colour and place it in the VIC register at 53280.
2) Obtain the value for the screen background colour and place it in the VIC register at 53281.

Colour — BASIC listing

```
15000 REM*******************************
15001 REM SCREEN AND BACKGROUND COLOUR
15002 REM*******************************
15010 INPUT "SCREEN BORDER COLOUR";X
15020 IF X<0 OR X>15 THEN PRINT "ILLEGAL
 VALUE" : GOTO 15010
15030 INPUT"SCREEN BACKGROUND COLOUR";Y
```

```
15040 IF Y<0 OR Y>15 THEN PRINT "ILLEGAL
 VALUE" : GOTO 15010
15050 POKE 53280,X
15060 POKE 53281,Y
15999 END
```

Colour — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 SYM
00                 30 ORG $C1B0
C1B0               70 VIC     = $D000
C1B0               80 GETPAR  = $C0FC
C1B0               90 IQERR   = $C1AA
C1B0   20FCC0     130 SETCOL JSR GETPAR
C1B3   E010       140 CPX #$10
C1B5   B0F3       150 BCS IQERR
C1B7   C010       160 CPY #$10
C1B9   B0EF       170 BCS IQERR
C1BB   8E20D0     180 STX VIC+$20
C1BE   8C21D0     190 STY VIC+$21
C1C1   60         200 RTS
C1C2              210 END

 TOTAL ERRORS IN FILE --- 0


VIC                D000
GETPAR             C0FC
IQERR              C1AA
SETCOL             C1B0
 TOTAL NUMBER OF SYMBOLS ---- 4
```

Machine code:

```
ADD    DATA                          CHECKSUM

C1B0   20 FC C0 E0 10 B0 F3 C0        7AE6
C1B8   10 B0 EF 8E 20 D0 8C 21        6039
C1C0   D0 60                          0200
```

Commentary

Line 130: The parameters for screen border and background colours are picked up by GETPAR (a subroutine of LOCATE which you have already entered) and stored in registers X and Y.

Lines 140–170: The two parameters are compared with 16 to ensure that they are valid colours.

Lines 180–190: The two colour values are stored in the relevant VIC registers.

Testing

BASIC: Entering the value one in response to both prompts should set the whole of the screen to white.

MACHINE CODE: Entering SYS (49584) 1,1 should set the whole of the screen to white.

Syntax

The syntax for the command is:

SYS (49584) { BORDER} ,{ BACKGROUND}

where both BORDER and BACKGROUND are numbers in the range 0–15.

6. PLOTTING A LOW RESOLUTION PIXEL (PLOT)

The purpose of this routine is to allow straightforward plotting of a low resolution pixel (ie one quarter of a character square) by means of the command PLOT. The *method* we shall have to use is not quite as straightforward, however. What we cannot do is simply to write a routine which will address all the 80*50 possible pixel positions on the screen. In low resolution mode, the 64 does not give us the option to adopt this simple solution.

The key to the solution of the problem lies in the fact that, whichever of the four possible pixels are set in any one character square, there is already a graphics character in the 64's graphics set which is capable of expressing that combination. In other words, with sufficient low cunning it *should* be possible to use existing characters, printed on to the screen, to simulate low resolution plotting. The particular characters in question are:



Using these characters, it will be possible to plot any combination of the quarters of any character square, or indeed to erase them.

The routines which follow are based on the convention that any of the 25*40 character squares on the 64's screen can be subdivided into four and the quarters numbered as follows:

0 1
2 3

To be able to switch any of these four blocks on or off will require a total of

16 different shapes (2^4) capable of going into the one character square.

The really important step is to assume that the numbers in the square illustrated above are powers of two. Once this is done it is possible to suggest a very logical way of looking at the 16 characters. Take the following character square:

```
0 1
1 0
```

Here the sections with values of 1 and 2 are filled in and if we regard these as powers of two then the 'value' of the character square is 6 (2^1 + 2^2). Take a look at the following character squares and make sure you understand why their values are as they are:

```
1 0     1 0     1 1     0 0     1 1     0 0
1 0     0 1     0 0     1 1     1 1     0 0
5       9       3       12      15      0 0
```

When the principle is properly understood, and provided the characters needed are in a logical order, it becomes simplicity itself to find the right character to fit the desired arrangement of pixels in a square.

Using powers of two also permits us to use the logical operators AND, OR and XOR to their fullest in setting and resetting individual pixels by means of the use of 'masks'. Masks are one of the most powerful techniques available to the machine code programmer and simple to use once they are understood. The purpose of a mask is to allow individual bits in a byte to be changed, or indeed preserved, regardless of what is being done to other bits.

One or two simple examples may serve to illuminate the issue. Suppose that it is desired to switch on an individual bit, say bit four, without affecting the rest of the bits within the byte. The standard technique, which we have already used several times, is to create a value where *only* that bit is set, to place it in the accumulator and then to OR it with the byte which we wish to change. In this case the value stored in the accumulator (16 = 2^4) was the mask. If we had wanted to switch off the individual bit, then we would have created a mask which had every bit set *except* the one to be operated upon, and then ANDed the mask with the byte to be changed. The mask necessary to switch off bit four would have been of a value of 255 – 16. Finally there is the case where it is desired to switch a bit to the opposite of its present state, regardless of what that is. Here a mask is created with only the relevant bit set but, instead of OR, XOR is used. 'Exclusive OR' ensures that, if the bit in the mask is set and that in the target byte is not, then the target bit is set: if both the mask bit and the target bit are set, then the target bit is reset. Using the two masks, then, in combination with OR, AND and XOR, we can home in on any bit, or even combination of bits, we wish. This ability will be crucial to the low-resolution

plotting routine.

The procedure for PLOT is as follows:

1) Obtain the coordinates for the low resolution pixel to be plotted.
2) Ascertain whether the pixel is to be set, reset or switched to the opposite of its present state — ie the MODE of the plotting to be done.
3) Create a mask representing the value of the single pixel within the character square, as calculated according to the method described above.
4) Find out the value, again in terms of the method of calculation described above, of what is currently on the screen at the specified character position.
5) Use OR, AND or XOR to combine the mask created with the value of what is currently on the screen.
6) Place the new character on the screen.

Plot — BASIC listing

```
16000 REM*******************************
16001 REM PLOT PIXELS
16002 REM*******************************
16010 DATA 32,126,124,226,123,97,255,236
,108,127,225,251,98,252,254,160
16011 REMDATA 32,97,98,108,123,124,126,1
27,160,225,226,236,251,252,254,255
16020 DIM PLOT (15)
16030 FOR I=0 TO 15 : READ PLOT(I) : NEX
T
16040 MASK=0
16045 PRINT "[CLR]"
16050 INPUT "[HOME]MODE:
    [CL][CL][CL][CL][CL][CL][CL][CL][CL]
[CL][CL][CL][CL][CL][CL][CL][CL][CL][CL]
[CL][CL]";MODE
16060 IF MODE <0 OR MODE>2 THEN 16050
16070 INPUT "[HOME]X CO-ORDINATE:
             [CL][CL][CL][CL][CL][CL][CL
][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL
][CL][CL][CL][CL]";X
16080 IF X<0 OR X>79 THEN 16070
16090 INPUT "[HOME]Y CO-ORDINATE:
             [CL][CL][CL][CL][CL][CL][CL
][CL][CL][CL][CL][CL][CL][CL][CL][CL][CL
][CL][CL][CL][CL]";Y
16100 IF Y<0 OR Y>49 THEN 16090
16110 X1=INT(X/2) : Y1=INT(Y/2)
16120 MASK=-(X/2<>INT(X/2)) - 2*(Y/2<>IN
T(Y/2))
16130 PC=PEEK(1024+40*Y1+X1)
```

```
16140 FLAG=-1 : FOR I=0 TO 15 : IF PC=PL
OT(I) THEN FLAG=I
16142 NEXT I
16145 IF FLAG<>-1 THEN PC=FLAG
16150 IF FLAG=-1 THEN PC=2^MASK
16160 IF MODE=0 THEN PC=PC OR 2^MASK
16170 IF MODE=1 THEN PC=PC AND (15-2^MAS
K)
16175 IF MODE=2 THEN PC=PC - 2^MA*((PC A
ND 2^MA)=0) + 2^MA*((PC AND 2^MA)=2^MA)
16180 POKE 1024+40*Y1+X1,PLOT(PC)
16190 POKE 55296+40*Y1+X1,14
16200 GOTO 16050
16999 END
```

Commentary

Lines 16010–16030: The values in the DATA statement are the screen codes of the 16 characters, in that order. They are placed into the array PLOT for later use.

Line 16040: The variable MASK will be used to hold the mask value.

Lines 16060–16110: The MODE parameter and the coordinates of the pixel are input. The MODE values are as follows:

0 = set pixel
1 = reset pixel
2 = invert pixel

The final line of this section derives the coordinates of the actual character position in which the pixel lies.

Line 16120: The method of calculating the value of the mask depends simply upon whether the coordinates, both horizontal and vertical, are odd or even. Both coordinates are numbered from zero so, if the horizontal coordinate is even, the pixel to be set is on the lefthand side of a character square, while, if the vertical coordinate is even, the position must be on the top line of a square.

Lines 16130–16150: The value of the current character in the specified screen position is obtained by the use of PEEK and compared with the set of quarter-square graphics characters. If it corresponds to one of them, its value will be stored in FLAG. If the character on the screen is not one of the existing character positions then it is taken to be an empty space, ie plotting over text will wipe out the character in that position.

Lines 16160–16175: The mask is applied with OR, AND or a simulation

of Exclusive OR.

Line 16180: The value for the new character is POKEd on to the screen.

Plot — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 SYM
00                 30 ORG $C1C2
C1C2              120 GETBYT = $C006
C1C2              130 GETPAR = $C0FC
C1C2              140 IQERR  = $C1AA
C1C2              150 MASK   = $FD
C1C2              160 MODE   = $FE
C1C2              170 DOAT   = $C10D
C1C2              180 COMMA    = $AEFD
C1C2  A900        210 PLOT  LDA #0
C1C4  85FD        220 STA MASK
C1C6  2006C0      230 JSR GETBYT
C1C9  C903        240 CMP #3
C1CB  B0DD        250 BCS IQERR
C1CD  85FE        260 STA MODE
C1CF  20FDAE      270 JSR COMMA
C1D2  20FCC0      290 JSR GETPAR
C1D5  8A          310 TXA
C1D6  4A          320 LSR A
C1D7  AA          330 TAX
C1D8  26FD        340 ROL MASK
C1DA  98          350 TYA
C1DB  4A          360 LSR A
C1DC  A8          370 TAY
C1DD  26FD        380 ROL MASK
C1DF  200DC1      400 JSR DOAT
C1E2  A6FD        420 LDX MASK
C1E4  BD3AC2      430 LDA MTAB.X
C1E7  A6FE        440 LDX MODE
C1E9  CA          450 DEX
C1EA  D002        460 BNE L000
C1EC  490F        470 EOR #$F
C1EE  85FD        480 L000 STA MASK
C1F0  A4D3        500 LDY $D3
C1F2  B1D1        510 LDA ($D1).Y
C1F4  A20F        520 LDX #$F
C1F6  DD2AC2      530 L001 CMP CHRTAB.X
C1F9  F003        540 BEQ L002
C1FB  CA          550 DEX
C1FC  D0F8        560 BNE L001
C1FE  8A          570 L002 TXA
C1FF  A6FE        590 LDX MODE
```

```
C201  F00D      600 BEQ SET
C203  CA        610 DEX
C204  F005      620 BEQ CLR
C206  45FD      640 EOR MASK
C208  4C12C2    650 JMP L003
C20B  25FD      670 CLR AND MASK
C20D  4C12C2    680 JMP L003
C210  05FD      700 SET ORA MASK
C212  AA        720 L003 TAX
C213  BD2AC2    730 LDA CHRTAB.X
C216  91D1      750 STA ($D1).Y
C218  A5D2      770 LDA $D2
C21A  A6D1      780 LDX $D1
C21C  2903      790 AND #$03
C21E  09D8      800 ORA #$D8
C220  85F4      810 STA $F4
C222  86F3      820 STX $F3
C224  AD8602    830 LDA 646
C227  91F3      840 STA ($F3).Y
C229  60        850 RTS
C22A  207E7C    870 CHRTAB BYT 32.126.12
4.226.123.97.255.236.108.127.225.251.98
C237  FCFEA0    880 BYT 252.254.160
C23A  010204    900 MTAB BYT %0001.%0010
.%0100.%1000
C23E            910 END

 TOTAL ERRORS IN FILE --- 0


GETBYT          C006
GETPAR          C0FC
IQERR           C1AA
MASK            FD
MODE            FE
DOAT            C10D
COMMA           AEFD
PLOT            C1C2
L000            C1EE
L001            C1F6
L002            C1FE
CLR             C20B
SET             C210
L003            C212
CHRTAB          C22A
MTAB            C23A
 TOTAL NUMBER OF SYMBOLS --- 16
```

Machine code

```
ADD   DATA                          CHECKSUM
```

```
C1C2  A9 00 85 FD 20 06 C0 C9   7851
C1CA  03 B0 DD 85 FE 20 FD AE   5C88
C1D2  20 FC C0 8A 4A AA 26 FD   75E1
C1DA  98 4A A8 26 FD 20 0D C1   7F23
C1E2  A6 FD BD 3A C2 A6 FE CA   B8EE
C1EA  D0 02 49 0F 85 FD A4 D3   7CC7
C1F2  B1 D1 A2 0F DD 2A C2 F0   ABF4
C1FA  03 CA D0 F8 8A A6 FE F0   6754
C202  0D CA F0 05 45 FD 4C 12   5E16
C20A  C2 25 FD 4C 12 C2 05 FD   933F
C212  AA BD 2A C2 91 D1 A5 D2   9F88
C21A  A6 D1 29 03 09 D8 85 F4   9236
C222  86 F3 AD 86 02 91 F3 60   A25A
C22A  20 7E 7C E2 7B 61 FF EC   5566
C232  6C 7F E1 FB 62 FC FE A0   8B2C
C23A  01 02 04 08               0020
```

Commentary

Lines 210–220: The value of the variable MASK is initially set to zero. Since the values for any one square can range from zero to fifteen, four bits may be needed, each of the four bits indicating whether one of the four pixels is set.

Lines 230–270: These lines use GETBYT to pick up the parameter for the mode. The modes are as in the BASIC version of the routine.

Line 290: GETPAR is used to pick up the two parameters for the pixel position, with 0,0 as the top lefthand corner of the screen and 79,49 as the bottom righthand corner.

Lines 310–380: Each parameter is divided by two, thus giving the coordinates of the correct character position on the screen. In addition, shifting the contents of the accumulator left, through the carry flag, will leave the carry flag set if the particular coordinate is an odd value. The one ROL instruction will transfer this set bit to the contents of MASK. Once both ROL instructions have been carried out, the position of the pixel within the character square is recorded in MASK.

Line 400: This call to part of the LOCATE routine provides a check that the character position so far derived is in fact on the screen and positions the cursor.

Lines 420–430: We have already noted that the most logical way to regard the values of the four positions within the character square is as powers of two. This will allow us to store the condition of each pixel in one unique bit

within MASK. At the moment, what we have is a representation according to the little square described in the introduction to this section. These lines use a table of values at the end of the routine to transform a value such as three into another where *bit* three is set. The reason for the transformation is that eventually we will be able to represent every possible state of the four pixels with values expressible in four bits but not with two. The method of achieving the transformation, using a table and accessing it by means of indexed addressing, may not be the most stylish way of achieving the end result, but it is considerably shorter for a small number of values than the coding necessary to transform the value without a table.

Lines 440–480: The mode parameter has been contained in location MODE since being picked up by GETPAR. It is now transferred to the X register and decremented by one. If the resulting value is either one or 255 the mode must have been either two or zero. For both of these modes, a mask with a single bit set is suitable, so no further action is taken. If, however, the resulting value is zero (ie the original value was one) then the mode requested was 'erase' and the mask value obtained in line 430 is inverted by the use of an exclusive OR with 15 and the result stored back in MASK.

Lines 500–510: The position at which a plot is being made has already been determined. The character at that position on the screen is picked up by these two lines and placed in the accumulator.

Lines 520–570: This is a loop which will potentially be repeated 15 times, comparing the values in the table CHRTAB at the end of the routine with the actual value of the character picked up from the screen. If the whole of the loop is executed without finding a character to match the one picked up from the screen, the value in X will have reached zero and the character picked up from the screen will be assumed to be a blank space.

Lines 590–600: MODE is tested to see if it is zero. If so, a branch is made to the part of the routine which finally sets up the character to be printed.

Lines 610–620: MODE is tested to see if it is one. If so, a branch is made to the routine which will clear one of the bits in the current character value.

Lines 640–650: To reach this point in the routine, MODE must equal two, the invert mode, and this can be easily accomplished by XORing of the four-bit character value in the accumulator with MASK.

Lines 670–680: These lines clear the relevant bit by ANDing the character

value with the value in MASK.

Line 700: The routine to set one bit.

Lines 720–730: The table of characters is now used in the opposite direction to translate our private character value back into a normal screen code.

Line 750: The new character is placed on to the screen.

Lines 770–850: The call to DOAT (part of locate) in an earlier line has already set the normal register, which records the position in memory of the print cursor, to the character which has to be changed. This value is loaded into the A and X registers. The most significant byte is ANDed with three, thus removing those bits which refer to the part of the address over and above its position in relation to the beginning of the screen. The high byte is now ORed with $D8, which makes the address contained in registers A and X now refer to the colour memory, with the position in colour memory corresponding to the current print position. The current print colour is now loaded from location 646 into the accumulator and transferred to the correct location in colour memory.

Lines 870–900: The tables which are used to set up the mask value and identify the various screen characters and translate them into the logically ordered sequence we have been using.

Testing

BASIC: Simply RUN the routine and enter a variety of values and MODEs to check that you can set, reset or invert a pixel at any position on the screen. Entering the following coordinates, all in MODE zero, should build up a character square in the centre of the screen:

```
40,20
40,21
40,20
40,21
```

MACHINE CODE: Clear the screen and then enter the following:

```
10  FOR MODE = 0 TO 2
20  FOR I = 20 TO 60 + 20*(MODE = 1)
30  SYS (49602) MODE, I,25
40  NEXT I, MODE
```

What you should see is a line drawn across the screen which is quickly half erased then, again quickly, the erased half is redrawn and the remaining

half erased.

Syntax

The syntax of the command is:

SYS (49602) { MODE} , { ROW} , { COLUMN}

where MODE is a value between 0 and 2 as follows:

0 — set pixel specified
1 — clear pixel specified
2 — invert pixel specified

ROW is a number in the range 0–49 with the top of the screen representing zero. COLUMN is a number in the range 0–79, with the lefthand side of the screen representing zero.

7. RESERVING MEMORY (ALLOT)

ALLOT is not really a graphics routine in itself, but a tool which makes possible graphics which require the setting aside of memory. The Commodore 64 has an almost unparalleled ability to reconfigure its memory, that is to say to rearrange the memory areas given over to specialised functions like the screen, the BASIC program, the character data and so on.

ALLOT's purpose is to allow the user to specify that a free area of memory is needed at the beginning of the memory (eg for sprite data or in order to move the screen) and then to shift the whole of the program *including the variables area* up the memory to provide the required space. ALLOT can be used while the program is running, thus doing away with the need to load a preliminary program to reconfigure the memory.

The procedure for ALLOT is as follows:

1) The number of bytes to be ALLOTted is obtained.
2) This number is added to the existing start of BASIC, *which may or may not be the normal start position at 2048.*
3) The program in memory is moved upwards by the number of bytes stipulated.
4) The various pointers which record important memory locations for the BASIC program are updated.

Allot — BASIC listing

As already hinted, this command cannot be properly simulated in BASIC. The reason for this is that the BASIC program would have to move itself *while it was being executed*, which would make it impossible for the 64's operating system to know where the next instruction in the BASIC pro-

gram was meant to be coming from.

The following small routine is one that can be run *before* the main program is loaded: it will simply reset the start-of-BASIC pointer to provide the desired free memory at the start of what used to be the program area. Note that, since the main program will be subsequently loaded, there is no need to update the rest of the BASIC pointers, they will be set correctly when the main program is loaded on top of the new start of BASIC.

```
21000 REM*******************************
21001 REM ALLOT SPACE AT PROG START
21002 REM*******************************
21003 REM NB THIS MUST BE RUN BEFORE
      LOADING THE MAIN PROGRAM
21004 REM*******************************
21010 INPUT "NEW START ADDRESS:";PS
21015 POKE PS,0 : PS=PS+1
21020 P2=INT(PS/256) : P1=PS-256*P2
21030 POKE 43,P1 : POKE 44,P2
21040 CLR
21999 END
```

Allot — assembly language listing

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $C23E
C23E              30 SYM
C23E              80 GETWRD = $C000
C23E              90 BSTART = $002B
C23E             100 BEND   = $0031
C23E  18         130 ADD CLC
C23F  B500       140 LDA 0,X
C241  6514       150 ADC $14
C243  9500       160 STA 0,X
C245  B501       170 LDA 1,X
C247  6515       180 ADC $15
C249  9501       190 STA 1,X
C24B  60         200 RTS
C24C  2000C0     230 ALLOT JSR GETWRD
C24F  A52B       240 LDA BSTART
C251  A42C       250 LDY BSTART+1
C253  38         260 SEC
C254  E901       270 SBC #1
C256  B001       280 BCS L000
C258  88         290 DEY
C259  855F       300 L000 STA $5F
C25B  8460       310 STY $60
C25D  A531       320 LDA BEND
C25F  A432       330 LDY BEND+1
```

```
C261   855A       340 STA $5A
C263   845B       350 STY $5B
C265   8558       360 STA $58
C267   8459       370 STY $59
C269   A258       380 LDX #$58
C26B   203EC2     390 JSR ADD
C26E   A8         410 TAY
C26F   A558       420 LDA $58
C271   20B8A3     440 JSR $A3B8
C274   A22B       460 LDX #BSTART
C276   203EC2     470 L001 JSR ADD
C279   E8         480 INX
C27A   E8         490 INX
C27B   E031       500 CPX #BEND
C27D   90F7       510 BCC L001
C27F   A27A       515 LDX #$7A
C281   203EC2     517 JSR ADD
C284   4C33A5     520 JMP $A533
C287               530 END

 TOTAL ERRORS IN FILE --- 0


GETWRD             C000
BSTART             2B
BEND               31
ADD                C23E
ALLOT              C24C
L000               C259
L001               C276
 TOTAL NUMBER OF SYMBOLS --- 7
```

Machine code

```
ADD    DATA                           CHECKSUM

C23E   18 B5 00 65 14 95 00 B5          4339
C246   01 65 15 95 01 60 20 00          2778
C24E   C0 A5 2B A4 2C 38 E9 01          9CF3
C256   B0 01 88 85 5F 84 60 A5          77FD
C25E   31 A4 32 85 5A 84 5B 85          562B
C266   58 84 59 A2 58 20 3E C2          66BE
C26E   A8 A5 58 20 B8 A3 A2 2B          93FB
C276   20 3E C2 E8 E8 E0 31 90          51F2
C27E   F7 A2 7A 20 3E C2 4C 33          BB03
C286   A5                              00A5
```

Commentary

Lines 130–200: A subroutine which will be commented upon later.

Line 230: GETWRD is used to pick up the number of bytes to be allotted.

Lines 240–250: The address of the current start of BASIC is obtained from the zero page and placed in the A and Y registers.

Lines 260–290: To ensure that the zero at the start of the program area is also moved, the start-of-BASIC pointer stored in registers A and Y is decremented by one. This is a simple example of sixteen-bit subtraction, with the one being first subtracted from the least significant byte, and then the carry flag tested to see if the high byte needs to be decremented because of a carry.

Lines 300–310: The contents of the A and Y registers are now transferred to a temporary storage location in zero-page memory, from where they will be picked up by the move routine.

Lines 320–370: Registers A and Y are now used to pick up the pointer to the end of the BASIC area and transfer this to two locations in zero page which will be used by the move routine.

Lines 380–390: Address $58–9 will be used to store the end-of-BASIC pointer. The ADD routine is called to add the contents of $14–15 (the amount of memory to be set aside) to this pointer, which now shows where the program will eventually end.

Lines 410–420: At the end of the ADD routine, the high byte of the new end of BASIC pointer is contained in the A register. This is now transferred to the Y register and the low byte loaded in to the A register.

Line 440: This is a call to the BASIC file editor. When a new line is being entered into the program, the file editor automatically tidies up the string memory, performs a check to see that there is enough room for the line and then moves the program up. Thus this one call performs the main part of the work for us.

Lines 460–510: The series of pointers at $2B–C (which are used by the system to identify the start of BASIC) variables and arrays are now incremented by the amount specified for the newly ALLOTted memory.

Line 520: This jump is to the ROM routine which 'rechains' the lines of the BASIC program — that is, recreates the link bytes attached to each line which are are used by the BASIC file editor system to point to the location of the next line in the memory. This is necessary since the lines have all been moved up by ALLOT.

Testing

BASIC: Enter the routine shown and then RUN it. ALLOT 1024 bytes when prompted. Now, without switching off the machine or pressing RUN/RESTORE, load another fairly simple program and test briefly that it runs properly. Stop the new program and enter in direct mode:

```
PRINT PEEK (43) PEEK (44)
```

The values displayed should be 0 and 12, indicating that the start address of BASIC is now 12*256, or 3072, exactly 1024 bytes more than the normal start position.

MACHINE CODE: To test the routine, install it in memory and then enter the following lines of BASIC:

```
10 A = FRE(0)
20 SYS(49740) 100
30 PRINT A-FRE (0)
```

The result should be the display of 100 on the screen (the amount of memory ALLOTted and therefore no longer available to BASIC). If the routine has been incorrectly entered it is likely that the system will crash.

Syntax and notes on use

It is important to use ALLOT with care since, once memory has been ALLOTted, there is no way to de-ALLOT it without switching off your 64. For this reason, you should take from BASIC only as much memory as you really need. You should also be careful to access the ALLOT command only once since it does *not* check to see whether memory has already been allotted, it merely ALLOTs the specified amount every time the command is carried out. To see the effect of this, add the following to the lines used for testing above:

```
40 PRINT FRE(0)
50 RUN
```

The program will print out a string of figures, each 100 less than the previous one, and will eventually crash with the OUT OF MEMORY error displayed. The reason for this is that, every time the program was RUN, it took another 100 bytes away from BASIC.

The problem can be avoided by replacing line 20 of the test program with the following and omitting lines 10 and 30 (you will have to turn the machine off and start again):

```
15 N = 100
20 T = PEEK(43) + 256*PEEK(44):IF T-2049< N THEN SYS (49740)
   N-T + 2049
```

Here, N is the number of bytes you wish to ALLOT and line 20 first checks to see that these bytes have not already been ALLOTted. If an ALLOT of at least the right size has already been performed, no further action is taken.

The syntax for ALLOT is:

SYS (49740) { AMOUNT OF MEMORY TO BE FREED}

If the amount of memory ALLOTted at one time is too great, then an OUT OF MEMORY error will be generated.

The command may be used during the course of a program, since no variables are lost in the process. One limitation, however, is that the DATA pointer is not updated when the program is moved, so a RESTORE *must* be executed if you wish to read data after ALLOT has been used.

Note: While ALLOT may be successfully used in direct mode (ie without a line number) it generates a SYNTAX ERROR message.

8. COPYING THE CHARACTER SET (CCOPY)

The 64 allows almost total flexibility to the user in redefining the character set. To make use of that flexibility, however, it is necessary to move all of the data which defines the characters in the first place from the ROM, where it cannot be altered, to a place chosen in RAM. In principle, this is straightforward and can be done in a few lines of BASIC, as this section shows. The drawback is the amount of time this takes.

The one slight oddity about the whole procedure is that the character data is not actually visible to the CPU in the normal course of events. If you look at the area of memory beginning at 53248, which is where character data is said to be held, you will find that, according to the normal memory map, it is occupied by the VIC chip among other things. This is partly because the CPU is not usually called upon to do anything with the character data — this is a job for the VIC chip — and partly to cram the maximum amount into the 64K of memory available. Fortunately, a temporary adjustment to the value of the register at address 0001 in memory serves to reconfigure the memory so that the character data is visible to the CPU, replacing the area occupied by the VIC chip, etc.

This does not totally solve the problem since the VIC chip is an integral part of the system and the area of memory used for its registers is constantly being scanned by the system for information about its communications with external devices. If we replace the usual registers with the character data, we could be giving the system useless information. The solution is, for the time that the character data occupies the memory from 53248 onwards, to switch off all the interrupts which are normally reminding the system to scan the input/output information. While these

interrupts are switched off, no communication will be possible with the 64 — it will be working in total isolation which can only be disrupted by switching it off and on again.

The procedure for moving the character set data into RAM is as follows:

1) Obtain the address at which the character set data is to be stored.
2) The input/output interrupts are switched off (see above) and the memory of the 64 reconfigured so that the character data is brought into sight of the CPU or 'mapped in'.
3) The 4K bytes of character data are copied from the ROM to an area of RAM specified by the user.
4) The character ROM is mapped out of the memory again and the interrupts switched back on.

You might like to note that the SHAPE command given in the sprite section can be adapted to redefine characters once the character data has been placed into RAM.

CCopy — BASIC listing

```
17000 REM*******************************
17001 REM MOVE CHARACTER SET
17002 REM*******************************
17010 INPUT "NEW ADDRESS:";NA
17020 IF NA/2048<>INT(NA/2048) THEN PRIN
T "NOT START OF 2K BLOCK" : GOTO 17010
17030 POKE 56334, PEEK(56334) AND 254
17040 POKE 1, PEEK(1) AND 251
17050 FOR I=0 TO 4095
17060 POKE NA+I, PEEK (53248+I)
17070 NEXT I
17080 POKE 1, PEEK(1) OR 4
17090 POKE 56334, PEEK (56334) OR 1
17999 END
```

Commentary

Line 17030: Resetting bit zero of the register at 56334 switches off the interrupts.

Line 17040: The character data is mapped into memory by resetting bit two of the register at 0001 in the memory.

Lines 17050–17070: This loop copies the data from the character ROM to the area specified by the user.

Lines 17080–17090: The character ROM is mapped out of memory and the interrupts turned back on.

CCopy — assembly language listing

```
ADD  DATA      SOURCE CODE
00               10 PRT
00               20 ORG $C013
C013             30 SYM
C013            110 GETBYT = $C006
C013            120 ADD    = $C23E
C013  2006C0    150 JSR GETBYT
C016  0A        152 ASL A
C017  0A        154 ASL A
C018  8515      158 STA $15
C01A  A900      160 LDA #$00
C01C  8514      165 STA $14
C01E  A0D0      170 LDY #$D0
C020  855F      180 STA $5F
C022  8460      190 STY $60
C024  A0E0      200 LDY #$E0
C026  855A      210 STA $5A
C028  845B      220 STY $5B
C02A  A010      230 LDY #$10
C02C  8558      240 STA $58
C02E  8459      250 STY $59
C030  A258      260 LDX #$58
C032  203EC2    270 JSR ADD
C035  AD0EDC    290 LDA 56334
C038  29FE      300 AND #254
C03A  8D0EDC    310 STA 56334
C03D  A501      320 LDA 1
C03F  29FB      330 AND #251
C041  8501      340 STA 1
C043  20BFA3    360 JSR $A3BF
C046  A501      380 LDA 1
C048  0904      390 ORA #4
C04A  8501      400 STA 1
C04C  AD0EDC    410 LDA 56334
C04F  0901      420 ORA #1
C051  8D0EDC    430 STA 56334
C054  60        440 RTS
C055            450 END

 TOTAL ERRORS IN FILE --- 0


GETBYT          C006
ADD             C23E
 TOTAL NUMBER OF SYMBOLS --- 2
```

Machine code

```
ADD   DATA                        CHECKSUM
C013  20 06 C0 0A 0A 85 15 A9      2057
```

```
C01B  00 85 14 A0 D0 85 5F E4   3796
C023  60 A0 E0 85 5A 84 5B A0   8286
C02B  10 85 58 84 59 A2 58 20   42A0
C033  3E C2 AD 0E DC 29 FE 8D   700D
C03B  0E DC A5 01 29 FB 85 01   58EF
C043  20 BF A3 A5 01 09 04 85   5F29
C04B  01 AD 0E DC 09 01 8D 0E   3CE4
C053  DC 60                     0218
```

Commentary

Line 60: The 1K block to which the ROM data is to be copied is obtained by use of GETBYT.

Lines 70–110: The 1K block address is converted to a full sixteen-bit address in $14–15.

Lines 120–140: The address of the first byte to be copied is placed into temporary storage in zero-page memory. The character generator ROM begins at $D000.

Lines 150–170: The same is done for the *last* byte to be copied from the ROM, $E000.

Lines 180–200: Finally, the number of bytes to be copied are stored, the value being $1000 or 4K (4096).

Lines 210–220: The ADD routine from ALLOT is used to add the number of bytes to be transferred to the start address specified by the user and to store the result in zero-page memory.

Lines 230–280: These lines turn off the input/output interrupts and tell the system to regard the character generator ROM as part of normal memory.

Line 290: A call to part of the BASIC file editor routine. The call is to the part of the routine which will move a block of memory but *after* the section which checks for sufficient space in memory, so that the data can literally be placed anywhere. Note that the BASIC program is not moved and so may be overwritten if you specify the wrong address to copy the character data to.

Lines 300–350: The character generator ROM is moved out of normal memory and the interrupts switched back on again.

Testing

BASIC: RUN routine and, when prompted, reply with the figure 8192. Having done this, wait until the program has finished, then NEW it and follow the procedure under b) below.

MACHINE CODE: a) Enter the following line of BASIC:

```
10  SYS (49171) 8
```

b) Enter the following line of BASIC:

```
20  FOR I = 8192 TO 8199 : PRINT PEEK (I) : NEXT
```

The first line moves the character data to the block beginning at address 8192 ($2000). The second line then prints out the eight data bytes for the first character which is '@'. The values should be:

```
60
102
110
110
96
98
60
0
```

Syntax and notes on use

The syntax for character move is:

SYS (49171) { ADDRESS}

where ADDRESS is a number between 0 and 63 representing a 1K block.

No check is made that you are not overwriting important data so it is necessary to be sure where your program is before moving the character data.

9. MOVING THE CHARACTER DATA POINTER (CHRPTR)

Just as the screen memory may be moved around within the current video bank, so may the location from which the VIC chip will take the data on which it bases the characters printed on the low resolution screen.

Changing the character pointer from its usual setting (which is to one of the two 2K areas beginning at 4096 and 6144 relative to the start of the video bank) means that the VIC chip sees not the shadow character data but whatever is genuinely in memory. The same happens if the video bank is changed to either one or three, since the shadow character sets are not present in those banks. Thus the CHRPTR command can be used to switch the attention of the VIC chip to different character sets which have been

previously saved in memory.

The procedure here is exactly the same as for the screen except that the character data must be stored on a 2K boundary rather than a 1K boundary for the screen and the position of the 2K area within the current video bank is controlled by the lower four bits of [SCREEN POINTERS] — the same register which controls the screen position.

ChrPtr — BASIC listing

```
14000 REM*******************************
14001 REM CHARACTER MOVE
14002 REM*******************************
14010 INPUT "ADDRESS";CA
14020 BANK=PEEK(648) AND 192
14030 IF INT(CA/16384)<>BANK THEN PRINT
"WRONG BANK" : GOTO 14010
14040 IF CA/2048<>INT(CA/2048) THEN PRIN
T "ILLEGAL ADDRESS" : GOTO 14010
14050 CA=CA-16384*INT(CA/16384)
14055 CA=CA/1024
14060 POKE 53272, (PEEK(53272) AND 240)
 OR CA
14999 END
```

ChrPtr — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                10 PRT
00                20 ORG *C180
C180              30 SYM
C180              70 VIC     = *D000
C180              80 GETBYT = *C006
C180   2006C0    110 CHRPTR JSR GETBYT
C183   0A        130 ASL A
C184   0A        140 ASL A
C185   4D8802    150 EOR 648
C188   29C0      160 AND #*C0
C18A   D011      170 BNE BANKER
C18C   A514      180 LDA *14
C18E   290F      190 AND #*F
C190   8515      200 STA *15
C192   AD18D0    210 LDA VIC+*18
C195   29F0      220 AND #*F0
C197   0515      230 ORA *15
C199   8D18D0    240 STA VIC+*18
C19C   60        250 RTS
C19D   4C9AC0    260 BANKER JMP *C09A
C1A0   000000    270 WRD 0.0.0.0.0
C1AA   4C48B2    280 IOERR  JMP *B248
```

```
C1AD                290 END

TOTAL ERRORS IN FILE --- 0


VIC                 D000
GETBYT              C006
CHRPTR              C180
BANKER              C19D
IQERR               C1AA
 TOTAL NUMBER OF SYMBOLS --- 5
```

Machine code

```
ADD   DATA                              CHECKSUM

C180  20 06 C0 0A 0A 4D 88 02   2CB6
C188  29 C0 D0 11 A5 14 29 0F   6569
C190  85 15 AD 18 D0 29 F0 05   67E9
C198  15 8D 18 D0 60 4C 9A C0   43E4
C1A0  00 00 00 00 00 00 00 00   0000
C1A8  00 00 4C 48 B2            0272
```

Commentary

Lines 110–170: The specified address is obtained and checked against the current video bank.

Lines 180–240: If it is in the correct bank, the address is altered as in lines 250–320 of SCREEN and placed into [SCREEN POINTERS].

Line 260: A jump to the BANK ERROR message.

Line 270: Spare bytes included for the purposes of the development of the routine — they must not be omitted.

Testing

BASIC: Ensure that you are in capitals mode and then RUN the lines and answer 6144 when prompted to give the new address. The result should be a change to lower case mode, since there is a second character set stored at that point which is lower case. RUN the program again and this time specify 12288. The screen should fill with garbage since the VIC chip is interpreting the random contents of the memory at that point as if it were character data. Pressing RUN/RESTORE will bring back the normal character set.

MACHINE CODE: Once again, ensure that you are in capitals mode and

then enter:

```
10 SYS (49536) 6
20 FOR I = 1 TO 1000 : NEXT
30 SYS (49536) 8
40 FOR I = 1 TO 1000 : NEXT
50 SYS (49536) 4
```

The effects of these lines should be to change the character display to lower case, then to garbage them back to the normal upper case character set.

Syntax

The correct syntax for CHRPTR is:

SYS (49536) { ADDRESS}

ADDRESS is a number in the range 0–63 which is within the current video bank and divisible by two.

CHAPTER 3
High Resolution Commands

1. UNDERSTANDING HIGH RESOLUTION

We now turn our attention to the high resolution capabilities of the 64, one of its strongest features and yet one of the more difficult to use from normal BASIC. With the commands you will enter during the next few sections, a variety of high resolution facilities will become part of your everyday programming, considerably extending the scope of the 64. Before considering the commands themselves, we shall take a look at the peculiarities of the high resolution screen and also at the way in which the VIC chip operates in high resolution mode.

Some home microcomputers on the market today normally work in the high resolution mode. What that means in practice is that they are capable of turning on or off any one of the tiny dots which you can see making up the picture if you look closely at the normal lettering on your screen. These dots are usually known as pixels, which is short for 'picture elements', and every computer display, whether it is called high or low resolution, is made up of them. The difference between high resolution and low resolution is not therefore the fineness of the detail in the picture you see on your television or monitor but the degree of control you have over that detail.

We have already seen, in the chapter on the low resolution screen, that the characters in normal low resolution text are made up of pixels which are as finely resolved as is possible for the 64. Clearly, the problem of high resolution is not a matter of the 64's inability to print a single pixel on the screen.

But now suppose that you were told to take a capital 'O' on the screen and to switch off some of the pixels to make it look like a 'C'. The simple answer is that you can't do it, at least in the 64's normal low resolution mode, and the reason behind the inability is the attempt to save memory.

Although the display that you see on your television is in as fine detail as the 64 is capable of, that detail is not stored in memory in the form of a record of which pixels are on and which are off. What actually happens is that an area of 1000 bytes is set aside to remember the contents of the screen from moment to moment and each of those 1000 bytes contains a value which is the screen code of a particular character. The display that you see on the screen is an entirely different matter. It is the product of the VIC

chip, which is constantly scanning the 1000 bytes of screen memory. Each time the VIC chip comes across a screen code in the screen memory, it examines the character generator memory and finds there the precise, pixel by pixel detail of the character to be printed and then puts that detail on to the screen.

The advantage of this method is that, instead of all the detail having to be stored for each of the 25*40 character positions on the screen, it can all be kept in a dictionary 2K long (in fact the 64 has two character sets and so reserves 4K for the information). To keep the complete detail for the whole of the screen would save nothing on the dictionary of character shapes but would mean that the amount of memory necessary to remember, pixel by pixel, what was on the screen, would be 8000 bytes, not 1000.

An example may help to make all this a little clearer. Let us suppose that your display consists of a clear screen except for a single letter 'A' in the top righthand corner. In low resolution mode, the VIC chip scans the 1000 bytes of screen memory and immediately comes across the value 1 in the first byte of that memory. Looking at the character generator memory, the VIC chip finds eight bytes in the position corresponding to character 1, as follows:

```
24
60
102
126
102
102
0
```

These values are meaningless as decimal numbers. We really only begin to see their significance when they are written one after another in binary:

```
00011000
00111100
01100110
01111110
01100110
01100110
00000000
```

It should take you only a moment to see that what we have here is a picture in ones and zeros of the letter 'A'. It is this picture, with the ones translated into pixels that is placed on to the screen. The shape of the letter 'A' has to be kept in memory somewhere, anyway, and using this method means that the complex shape of 'A' only needs a single byte in the screen memory. The limitation is that, while you can change 'A' for any other character in

the character set, you cannot change the shape of 'A' itself, nor print anything that is *not* part of the normal character set.

Let's turn now to high resolution. Here, whatever detail you see on your screen is no more than a picture of *exactly* what is in screen memory. To print the letter 'A' in high resolution mode (not that you will be able to until you've read further in this book) would need, somewhere in the screen memory, the bytes which go to make up 'A' which we saw in the little binary picture above. On finding these bytes, the VIC chip treats them in much the same way as it treated the details in the character generator memory when we were in low resolution mode. Every bit which is set will be translated into a pixel on the screen. We could no doubt set up 'A' on the screen but it now requires eight bytes and not one. In all, the high resolution screen would be capable of holding exactly the same amount of characters as the low resolution screen, but with the additional capability that anywhere within its grid of 320 pixels across and 200 pixels down, any single pixel can be accessed.

A great deal of fuss is made about the high resolution capabilities of different microcomputers. The fact is that, as with everything else in computing, it is a trade-off. To get the kind of spectacular screen displays which high resolution allows, you have to sacrifice something like one-fifth of your available memory on the 64. On machines with smaller memories the sacrifice is, of course, proportionately greater. For the majority of useful programs, the high resolution capability will add little or nothing apart from window-dressing and the memory lost may make many useful applications impossible. The good thing about the 64 is that it allows you the choice of whether you want to make the sacrifice or not. If you genuinely need to be able to draw lines and circles and switch on and off individual pixels, then high resolution mode is available. For most of the time you can bask in the amount of memory you have available and make do with the 64's outstanding low resolution character set.

The organisation of the high resolution screen

You will probably already have gathered, from what has been said above, that using the high resolution screen is not entirely straightforward. Unlike the low resolution screen, it requires constant thought as to where information is to be placed and how that place is to be accessed. The first problem to be solved is the organisation of the high resolution screen memory.

The high resolution screen memory, as we have already mentioned, requires a nominal 8K of memory to be set aside, although the screen itself uses only 8*1000 bytes, leaving 192 bytes unaccounted for. From that, it is a simple matter to deduce that, if the screen memory begins at 8192 in memory (more about where to put it later), then the top lefthand corner of the screen is dealt with by the byte at 8192 and the bottom righthand corner by the byte at 16191. So far, so good. The problem begins when we begin to

try and see which part of memory controls which part of the screen in between the start and the finish.

In fact, the high resolution screen is set up so that it simulates as closely as possible the structure of the low resolution screen. Although every pixel is addressable, the screen is still divided effectively into 1000 'character locations', with each of those locations having eight bytes.

Let us for the moment forget the starting address of the screen memory and assume that it starts at zero. In this case the first four character locations on the screen use the first 32 bytes of memory as follows:

[BYTE 00]	[BYTE 08]	[BYTE 16]	[BYTE 24]
[BYTE 01]	[BYTE 09]	[BYTE 17]	[BYTE 25]
[BYTE 02]	[BYTE 10]	[BYTE 18]	[BYTE 26]
[BYTE 03]	[BYTE 11]	[BYTE 19]	[BYTE 27]
[BYTE 04]	[BYTE 12]	[BYTE 20]	[BYTE 28]
[BYTE 05]	[BYTE 13]	[BYTE 21]	[BYTE 29]
[BYTE 06]	[BYTE 14]	[BYTE 22]	[BYTE 30]
[BYTE 07]	[BYTE 15]	[BYTE 23]	[BYTE 31]

Each of the bytes will control eight pixels on the screen, so that the first 64 pixels, as far as the memory is concerned, go as follows:

[00]	[01]	[02]	[03]	[04]	[05]	[06]	[07]
[08]	[09]	[10]	[11]	[12]	[13]	[14]	[15]
[16]	[17]	[18]	[19]	[20]	[21]	[22]	[23]
[24]	[25]	[26]	[27]	[28]	[29]	[30]	[31]
[32]	[33]	[34]	[35]	[36]	[37]	[38]	[39]
[40]	[41]	[42]	[43]	[44]	[45]	[46]	[47]
[48]	[49]	[50]	[51]	[52]	[53]	[54]	[55]
[56]	[57]	[58]	[59]	[60]	[61]	[62]	[63]

The second 64 pixels, numbered from 64 to 127, will form the next character block to the right, and so on.

This arrangement will continue to the end of the first line of the screen and then begin the second line with the fortieth character block, this block containing bytes 320–327.

From all of this, it can be seen that there are three problems to be solved when addressing a single pixel on the screen:

1) Which character square is to be addressed.
2) Which byte is to be addressed within the character square.
3) Which bit is to be addressed within the byte.

Having broken down the problem in this way, it becomes clear that all that is required is a series of very simple calculations. (*Note:* All the calculations are based on the assumption that coordinates, both high and low resolution are numbered from zero.)

1. Which character square

First of all, the pixel to be addressed must be defined in terms of its high resolution coordinates, ie its place in terms of the total 320*200 grid which the high resolution screen gives access to. Having defined this as, for instance, 160 across by 100 down, a position in the centre of the screen, the procedure is:

a) Divide the vertical coordinate by eight.
b) Divide the horizontal coordinate by eight.

The result is a new pair of coordinates which express the position of the character square in low resolution terms. In the case of our example, the character square containing pixel 160,100 is 20,12.

2. Finding the byte

We have already seen that each complete character square is made up of eight bytes. We can now find the number of the first byte in the relevant character square by the following method:

1) Multiply the low resolution vertical coordinate by 320. In the case of the example, the result is 3840.
2) Add to this the low resolution horizontal coordinate multiplied by eight. The example result is 3840 + 160 = 4000.
3) Now take the high resolution vertical coordinate and subtract from it eight times the low resolution vertical coordinate. Add the result to the figure arrived at in the first two steps.

Final result for the example is 4000 + (100 − 8*12) = 4004. The byte containing pixel 160,100 is number 4004 relative to the start of the screen memory.

3. Finding the bit

To calculate the correct bit within the relevant byte, take the high resolution horizontal coordinate and subtract from it the low resolution horizontal coordinate multiplied by eight. In the case of the example, the result is 160−8*20 = 0. We have now arrived at the position of the exact bit in memory to control the desired pixel *but* there is one small complication. In all of this we have assumed, as much for the sake of sanity as anything else, that the bits are numbered from zero to seven left to right. Unfortunately, as you will already know if you are familiar with machine code, the individual bits within the byte are numbered from right to left. *Don't panic.* All that needs to be done to rescue the situation is to subtract the bit number arrived at above from seven. Thus, in the case of our example, the exact byte and bit to control pixel 160,100 is:

BYTE 4004

BIT 7 – 0 = 7

The complete formula for all of this, expressed in BASIC is:

BIT NUMBER = 7 – (X AND 7)
BYTE NUMBER = SS + 320*INT(Y/8) + 8*INT(X/8) + (Y AND 7)

where X is the high resolution horizontal coordinate, Y is the high resolution vertical coordinate, and SS is the start address of the high resolution screen in memory.

The high resolution screen and the system

In this section, we shall briefly examine the relationship between the high resolution screen and the rest of the 64, including the VIC chip.

1. Entering and leaving high resolution mode

This is controlled by bit five of the VIC register at 53265 ($D011). When the bit is set, the 64 is in high resolution mode. This register will be described as [VIC CONTROL I] in the sections that follow.

2. Position of screen memory

The position of the high resolution screen within the current video bank is controlled by bit three of the register [SCREEN POINTERS]. When the bit is set, the high resolution screen will start at address 8192 relative to the start of the current video bank. When the bit is reset, the screen starts at zero relative. Note that to start and clear the screen from zero in bank zero will interfere with page zero memory and crash the system — the screen would be unusable anyway since the character sets will appear on the screen.

3. Colour memory

The high resolution screen makes no use of the normal colour memory area at 55296–56295. Immediately high resolution mode is entered, the VIC chip switches its attention to the 1K block defined by the upper four bits of the VIC register which normally indicates the position of the low resolution screen within the current video block, ie the register [SCREEN POINTERS]. The effect of the colour memory is to define the foreground and background colours for each character block on the high resolution screen. The upper four bits for each byte from 1024 to 2023 (if the colour memory is in its normal position) will determine the colour of any set pixels in the corresponding character square on the high resolution screen. The lower four bits will define the background colour for the square.

The colour memory may be moved around within the current video bank (taking the sprite pointers with it, by the way). Problems will, however, be experienced if the colour memory is located in the same place as character

data in banks zero or two, or within the screen memory itself. Moving the colour memory away from 1024 can be advantageous in that it allows the high resolution colour memory to be manipulated without corrupting the contents of the low resolution screen. In video bank zero, it is only possible to move the colour memory away from 1024 if the start of BASIC is first redefined above the new top of colour memory. In other banks this will not be a problem.

4. Screen editor pointer

We have already seen, when looking at the low resolution screen, that the system register at 648 is a vital pointer when it comes to changing the position of the screen. Unless it is changed to indicate the new screen position, actually manipulating a low resolution screen which has been moved from 1024 will be impossible. In the case of moving the colour memory away from 1024, this is not so, since it is only the VIC chip which really *needs* access to the high resolution colour memory. There are, however, advantages in moving the [EDITOR] pointer to follow the colour memory — a number of ROM routines can be used to help with moving the cursor around the colour memory to correspond to the position of the cursor on the high resolution screen.

2. SETTING UP THE HIGH RESOLUTION SCREEN

In this section, we shall examine three separate routines which will deal with the problems of turning the high resolution screen on, turning it off again, and what to do if the 64 returns to the normal edit mode, say at the end of a program or when an error message is generated.

Part 1. Set lo-res (LOW)

Before we go about turning the high resolution screen on, it is wise to enter this routine, which allows you to turn it off again, thus avoiding the problems of getting back to the normal mode for editing or input.

The high resolution screen is turned on and off by the setting and resetting of bit five of the VIC register at 53265 [VIC CONTROL I]. In addition to resetting this bit, however, there is also the need to ensure that [SCREEN POINTERS] and [EDITOR] point to the correct position for the low resolution screen. The reason that this is necessary is that, in the high resolution routines that follow, instead of using the normal low resolution screen memory as the colour memory for the high resolution screen, the high resolution colour memory is moved. This MOVE allows changes in the high resolution screen to proceed without corrupting the contents of the low resolution screen. In addition, [EDITOR] is altered to point to the new colour memory so that existing routines within the ROM can be used to carry out a number of functions.

The procedure for LOW is therefore:

1) Check that the high resolution screen is on before executing the rest of the routine. There is good reason for this check, which is a departure from the practice we normally adopt for short commands (executing them whether they are needed or not). In the case of LOW, variables are picked up to indicate where the low resolution screen was positioned before high resolution mode was called for. If the variable locations are used when high resolution has *not* been entered, they will contain meaningless data and might crash the system.
2) Turn off high resolution mode.
3) Ensure that the screen editor pointer is directed towards the same point in memory as the screen position indicator for the VIC chip.

Low — BASIC listing

(*Note:* The BASIC routines which simulate high resolution commands, with the exception of LINE and CIRCLE, are designed to be built up into a single program, since many of the modules work together. Subsequent routines should be added to this one.)

```
23000 REM*******************************
23001 REM TURN OFF HI-RES
23002 REM*******************************
23010 IF (PEEK(53265) AND 32)=0 THEN 230
99
23020 POKE 53272,BS
23030 POKE 53265,PEEK(53265) AND 223
23040 SE=BS AND 240
23050 SE=SE/4
23060 TT=PEEK(648) AND 192
23070 TT=TT + SE
23080 POKE 648,TT
23099 END
```

Commentary

Line 23010: Ensure that system is in high resolution mode before continuing.

Line 23020: The reinstatement of the screen position, as recognised by the VIC chip, to that prevailing when low resolution was quit.

Line 23030: Switch off high resolution.

Lines 23040–23080: The low resolution screen position is translated from

the 1K units held in the upper half of 53272 to the 256-byte units which are used by the editor pointer at 648: the editor pointer is updated without disturbing the two highest bits which record the video bank.

Low — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C700
C700               30 SYM
C700               70 TEMP    = $02DD
C700               80 VIC     = $D000
C700              110 LORES
C700   AD11D0     130 LDA VIC+$11
C703   2920       140 AND #$20
C705   F023       150 BEQ L000
C707   ADDD02     160 LDA TEMP
C70A   8D18D0     170 STA VIC+$18
C70D   AD11D0     190 LDA VIC+$11
C710   29DF       200 AND #$DF
C712   8D11D0     210 STA VIC+$11
C715              230 UPDATE
C715   AD8802     240 LDA 648
C718   29C0       250 AND #$C0
C71A   8D8802     260 STA 648
C71D   AD18D0     270 LDA VIC+$18
C720   29F0       280 AND #$F0
C722   4A         290 LSR A
C723   4A         300 LSR A
C724   0D8802     310 ORA 648
C727   8D8802     320 STA 648
C72A              330 L000
C72A   60         340 RTS
C72B              350 END
 TOTAL ERRORS IN FILE ---- 0


TEMP              2DD
VIC               D000
LORES             C700
UPDATE            C715
L000              C72A
 TOTAL NUMBER OF SYMBOLS ---- 5
```

Machine code

```
ADD    DATA                              CHECKSUM

C700   AD 11 D0 29 20 F0 23 AD    7D03
C708   DD 02 8D 18 D0 AD 11 D0    8C46
C710   29 DF 8D 11 D0 AD 88 02    6936
C718   29 C0 8D 88 02 AD 18 D0    6264
```

```
C720  29 F0 4A 4A 0D 88 02 8D    6179
C728  88 02 60                   0284
```

Commentary

Lines 130–150: The high resolution mode is entered by setting bit five of the VIC register at $D011 (53265). These lines test to see whether it is already set, ie whether the 64 is already in high resolution mode. If so, a jump is made to the end of the routine and no further action is taken.

Lines 160–170: The value stored in the memory location TEMP (a spare byte at 733) represents the position of the screen and character data within the current 16K video block before high resolution was turned on. Replacing this in the VIC register at $D018 means that, when we return to low resolution, the VIC chip will know where to look for the screen and character data, this register having been used for other purposes while high resolution mode was switched on.

Lines 190–210: ANDing the contents of the register at $D011 with 235 ensures that bit five is reset without any change to the rest of the register.

Lines 240–340: During the high resolution mode, [EDITOR] has been used to point to the start of the high resolution colour memory and must now be reset to point to the low resolution screen.

Lines 240–260: Remove from [EDITOR] all but the reference to the current 16K video block while the remainder of the lines take the screen position which has already been replaced in $D018 and add it to the 16K block value in 648. The 'shift' instructions are necessary, because the screen address within the current video block is stored in the upper bits of the register at $D018, but will be the lower bits in 648.

Part 2. Automatic TurnOf (TURNOF)

One of the problems of using the high resolution mode is that there are times when you want it to turn off without being told specifically to do so. If, for instance, you are using high resolution during the course of a program and an error is generated so that the program stops, you are in some difficulty if the 64 remains in high resolution mode since you will not be able to see the error message, edit any lines or input new lines until you get back to the normal low resolution or text screen.

The object of the routine that follows is to change the sequence of actions which the 64 normally goes through before printing READY on the screen. This is known as the 'warm start' routine and the only change that we make to it is to alter the 'vector' or pointer to where it starts, so as to make it execute the low resolution routine given previously. Now, whenever the program stops, the 64 itself will check that it is back in low

resolution mode. Note that the routine will have *no* effect until the changes are made by the next routine to the BASIC vector table.

There is no attempt to duplicate this routine in BASIC as it is itself simply a jump to a machine code routine.

TurnOf — assembly language listing

```
ADD.   DATA       SOURCE CODE
00                  10 PRT
00                  20 ORG $C72B
C72B                30 SYM
C72B                70 WARM    = $A483
C72B                80 LORES   = $C700
C72B                90 WVEC    = 770
C72B               110 TURNOF
C72B   2000C7      120 JSR LORES
C72E   A9A4        130 LDA #WARM/256
C730   A283        150 LDX #WARM-WARM/256*2
56
C732   8D0303      160 STA WVEC+1
C735   8E0203      170 STX WVEC
C738   4C83A4      180 JMP WARM
C73B               190 END
 TOTAL ERRORS IN FILE --- 0

WARM               A483
LORES              C700
WVEC               302
TURNOF             C72B
 TOTAL NUMBER OF SYMBOLS --- 4
```

Machine code

```
ADD    DATA                          CHECKSUM

C72B   20 00 C7 A9 A4 A2 83 8D        3CAB
C733   03 03 8E 02 03 4C 83 A4        1712
C73B                                  0000
```

Commentary

Line 120: This calls up the low resolution routine in Part 1 of this section.

Lines 130–170: The correct address of the starting point of the warm start routine is loaded into registers A and X. This correct address, which was altered when high resolution mode was entered (see next routine), is replaced into the BASIC vector table.

Line 180: Since we are now back in low resolution mode and the vector table is in its normal state, a jump can be made to the normal warm start routine.

Part 3. High resolution mode (HIGH)

We now turn to what should logically have been first, entering high resolution mode. In essence this is simply a matter of setting bit five in the register at $D011. As usual, however, there are a number of complications if we wish not simply to be able to enter high resolution mode but to *use* it. There is the matter of telling the VIC chip where the 8K high resolution screen is in the memory, and also in the colour memory. If we wish to return to low resolution mode after our excursion, we need to make some record of where the lo-res screen actually is. There is no point in assuming that it will always be in the start-up position at 1024, or we should not be able to use high resolution effectively if we moved the lo-res screen.

The procedure for entering high resolution mode is therefore as follows:

1) Check that the system is not already in high resolution mode in order to avoid storing nonsense data in the variables which record the low resolution screen position.
2) Tell the VIC chip where to find the high resolution screen.
3) Enter high resolution mode.
4) In the machine code version, change the jump table so that the high resolution screen will always be switched off if the machine goes into direct mode (ie the program stops).
5) Update the screen editor pointer.

High — BASIC listing

```
22000 REM*******************************
22001 REM TURN ON HI-RES
22002 REM*******************************
22010 IF (PEEK(53265) AND 32) THEN 22500
22020 BS=PEEK(53272)
22050 POKE 53272,PEEK(53272) OR 8
22060 POKE 53265,PEEK(53265) OR 32
22500 GET T$ : IF T$="" THEN 22500
```

Commentary

Line 22010: The routine is not executed if the system is already in high resolution mode.

Line 22020: BS is the record of the position of the low resolution screen within the current video block.

Line 22050: Sets up the new screen position at 8192. In the machine code version this address will be provided by another command.

Line 22060: Switch on high resolution mode.

High — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C73B
C73B               30 SYM
C73B               70 TEMP   = $02DD
C73B               80 VIC    = $D000
C73B               90 UPDATE = $C715
C73B              100 WVEC   = 770
C73B              110 TURNOF = $C72B
C73B              140 HIRES
C73B   AD11D0     160 LDA VIC+$11
C73E   2920       170 AND #$20
C740   D021       180 BNE L000
C742   AD18D0     190 LDA VIC+$18
C745   8DDD02     200 STA TEMP
C748   ADDE02     210 LDA TEMP+1
C74B   8D18D0     220 STA VIC+$18
C74E   AD11D0     230 LDA VIC+$11
C751   0920       240 ORA #$20
C753   8D11D0     250 STA VIC+$11
C756   A9C7       270 LDA #TURNOF/256
C758   A22B       280 LDX #TURNOF-TURNOF/2
56*256
C75A   8D0303     290 STA WVEC+1
C75D   8E0203     300 STX WVEC
C760   4C15C7     320 JMP UPDATE
C763              330 L000
C763   60         340 RTS
C764              350 END
 TOTAL ERRORS IN FILE --- 0

TEMP              2DD
VIC               D000
UPDATE            C715
WVEC              302
TURNOF            C72B
HIRES             C73B
L000              C763
 TOTAL NUMBER OF SYMBOLS --- 7
```

Machine code

```
ADD    DATA                          CHECKSUM

C73B   AD 11 D0 29 20 D0 21 AD        7C7F
C743   18 D0 8D DD 02 AD DE 02        63F2
C74B   8D 18 D0 AD 11 D0 09 20        754A
C753   8D 11 D0 A9 C7 A2 2B 8D        78F3
C75B   03 03 8E 02 03 4C 15 C7        1659
C763   60                             0060
```

Commentary

Lines 160–180: If the high resolution screen is already on, then these lines default around the routine. Trying to set the high resolution screen when it was already functioning would result in spurious values for the position of the lo-res screen being stored.

Lines 190–200: The contents of the [SCREEN POINTERS] register are now stored in address TEMP, so that the low resolution configuration can be easily restored once we have finished with high resolution.

Lines 210–220: The position within the current 16K block of the 8K high resolution and its separate colour memory have been stored in address TEMP + 1. At the moment, this will have to be done by a BASIC POKE, as described under the testing section. Later we shall add a new command to set up the location of the high resolution screen and colour memory. The value is now stored in the VIC register at $D018 to tell the VIC chip where the screen and colour memory will be.

Lines 230–250: These lines set the fifth bit of [VIC CONTROL I], thus entering high resolution mode.

Lines 270–300: You will remember, from the automatic turnoff routine earlier, that the warm start vector for BASIC needs to be altered so that high resolution is switched off every time the program stops. These lines alter the vector for the warm start routine to point to the auto-turnoff.

Line 320: While we are in high resolution it will be of some advantage to move the colour memory away from its normal location where the lo-res screen usually is. This will mean that any changes to the colour memory will not corrupt the text on the lo-res screen. Having moved the colour memory it will also be useful to have the screen editor functions directed towards the location of the colour memory rather than towards the low resolution screen, since this will allow us to use certain ROM functions in interesting ways while in high resolution mode. This line uses the UPDATE section of the LOW command to set [EDITOR] so that it will point to the new location of the colour memory.

Testing

BASIC: RUN the program (both sections in memory at once) and the screen will be filled with garbage. Press any key and the screen should return to normal. The garbage screen was the high resolution screen interpreting the random contents of the memory between 8192 and 16383.

MACHINE CODE: The only effective way to test these routines is, first of

all, to see that all three are entered. Starting with the high resolution command means that you become 'blind' to anything the system places on to the lo-res screen. Having entered all three of the routines, they can be tested by the following BASIC program:

```
 5 POKE 734,56
10 SYS 51003
20 FOR I = 1 TO 1000 : NEXT
30 SYS 50944
```

On running the program, you should see the screen immediately change to a total mess (unless you have previously cleared the high resolution memory since switching the machine on). After a pause, the normal screen will return. The pattern of dots and dashes which appears is the high resolution screen interpreting the random contents of the memory it is using. The blocks of colour are the characters of the low resolution screen being interpreted as colour attributes of individual character squares on the high resolution screen.

Syntax and use of HIGH and LOW

The correct syntax for the commands in this section is:

```
HIGH RESOLUTION ON : SYS 51003
HIGH RESOLUTION OFF : SYS 50944
```

The automatic turnoff routine is not designed to be called from BASIC.

In using the routines, the following limitations should be observed:

1) The location TEMP + 1 should not be used for any other purpose while the high resolution commands are in use since this will result in the VIC chip receiving spurious information on the location of the high resolution or low resolution screens.
2) The low resolution command should *only* be used to turn off the high resolution screen if it has been turned on with the routine from this section. If the high resolution screen has been turned on by any other method, the screen turnoff will pick up a nonsense value from the register TEMP + 1.
3) Changing the value in TEMP + 1 will not relocate the high resolution screen unless you follow this sequence:

   Change TEMP + 1 : call Lo-Res : call Hi-Res

   This is because high resolution will not act if the high resolution screen is already on.
4) The high resolution screen should not overlap any area of memory which the 64 uses for character data.
5) The high resolution screen must start on an 8K boundary within the

current 16K video block.

6) Colour memory must start on a 1K boundary.
7) Avoid hitting RUN/RESTORE while a high resolution routine of any kind is being executed. RUN/RESTORE interrupts the CPU itself and so will defeat the precautions taken to ensure that TURNOF returns the screen to BASIC. If you need to stop something, use the RUN/STOP key and then clear the screen with SHIFT/CLR. If you *do* accidentally use RUN/RESTORE you will need to enter (although you will not see it come up on the screen):

   POKE 648,4

   which will restore [EDITOR] to its start-up value.
8) One drawback to the high resolution commands is that, although they return the screen to low resolution if an error is encountered, they only do this *after* the error message has been printed on the colour memory area, which may not be the same as the low resolution screen. If you are encountering an error which you need to identify, either:

   a) do not move the colour memory away from its usual low resolution screen location, or

   b) run the routine, using all the high resolution commands but place REM in front of the lines which call up the high resolution screen and HCLEAR it. All the high resolution commands will proceed, even though you cannot see the high resolution screen. The low resolution screen will be corrupted by the process, but any error message will eventually be printed clearly.

9) In normal use you should always use the HSCREEN command (see next section) before calling up high resolution mode. This will ensure that the screen does not overwrite an unintended part of memory such as zero page.

3. SETTING UP SCREEN PARAMETERS (HSCREEN)

Now that we have the capability to set up the high resolution screen, we can profitably move on to the problem of defining its location in memory, together with the location of its associated colour memory. To set the location for the colour memory, it may be necessary to use routines like ALLOT to ensure that there is an area of memory available.

You might like to note that, for the sake of variety as much as anything else, we have chosen to allow the user to input the addresses for screen and colour in full, rather than in 1K block form. This gives an appreciation of a different range of techniques. If you really must have consistency you should find little difficulty in lifting the necessary routines out of the low resolution section to allow 1K block addresses to be used.

The procedure for HSCREEN is:

1) Get the screen address and check to see that it is in the correct bank.
2) Get the colour memory address and check that it is in the correct bank.
3) Convert the colour memory address into the four most significant bits for later storage in the VIC register at 53272.
4) Convert the screen memory address into the four least significant bits for later storage in the same register.
5) Combine the two values into a single byte.
6) Store the result in the byte at TEMP + 1 for later use by the HIGH command routine.

Since this routine works only with the machine version of HIGH, and has only two simple functions, a BASIC version is not provided.

HScreen — assembly language version

```
ADD.   DATA      SOURCE CODE
00                10 PRT
00                20 ORG *C764
C764              30 SYM
C764              80 TEMP    = *02DD
C764              90 GETWRD  = *C000
C764             100 COMMA   = *AEFD
C764             130 SETSCN
C764   2000C0     140 JSR GETWRD
C767   A515       160 LDA *15
C769   48         170 PHA
C76A   208BC7     180 JSR BANK
C76D   20FDAE     200 JSR COMMA
C770   2000C0     210 JSR GETWRD
C773   208BC7     220 JSR BANK
C776   A515       230 LDA *15
C778   0A         240 ASL A
C779   2A         250 ROL A
C77A   29F0       260 AND #*F0
C77C   8DDE02     270 STA TEMP+1
C77F   68         280 PLA
C780   4A         290 LSR A
C781   4A         300 LSR A
C782   290F       310 AND #*F
C784   0DDE02     320 ORA TEMP+1
C787   8DDE02     330 STA TEMP+1
C78A   60         340 RTS
C78B              370 BANK
C78B   A515       380 LDA *15
C78D   4D8802     420 EOR 648
C790   29C0       430 AND #*C0
C792   F003       440 BEQ L000
C794   4C9AC0     450 JMP *C09A
C797              460 L000
```

```
C797  60          470 RTS
C798              480 END
 TOTAL ERRORS IN FILE --- 0

TEMP              2DD
GETWRD            C000
COMMA             AEFD
SETSCN            C764
BANK              C78B
L000              C797
 TOTAL NUMBER OF SYMBOLS ---- 6
```

Machine code

```
ADD   DATA                              CHECKSUM

C764  20 00 C0 A5 15 48 20 8B    34E3
C76C  C7 20 FD AE 20 00 C0 20    98A0
C774  8B C7 A5 15 0A 2A 29 F0    8F6A
C77C  8D DE 02 68 4A 4A 29 0F    8899
C784  0D DE 02 8D DE 02 60 A5    4F6D
C78C  15 4D 88 02 29 C0 F0 03    350B
C794  4C 9A C0 60                06A8
```

Commentary

Lines 140–180: The first parameter, the start address of the screen, is picked up by GETWRD and passed to a subsequent section where it is checked for compatibility with the current bank.

Lines 200–220: The second parameter, for the start of colour memory, is similarly checked.

Lines 230–310: We have already noted that the VIC register at $D018 will contain two values within the 16K block, the colour memory address and the screen address. The four most significant bits will contain a record of which 1K block is occupied by the colour memory and the four least significant bits the record of the position of the high resolution screen. These lines manipulate the two specified addresses into the correct positions in the temporary register TEMP + 1. Note that only bit three is of any relevance in defining the position of the high resolution screen, which can *only* be located at zero or 8K in the current bank.

Lines 380–470: The function of these lines is to test that the two most significant bits of the parameters picked up are the same as those in 648, [EDITOR].

Testing

MACHINE CODE: The simplest test of the routine is to replace line 5 of the BASIC test routine you used for the high resolution commands in the

last section with:

```
5 SYS (51044) 8192, 3072
```

If you run the routine, you should now find that it works exactly as before — provided, of course, that you have all four of the high resolution routines in memory.

You can test the current routine in direct mode by entering:

```
POKE 734,0 : SYS (51044) 8192,3072 : PRINT PEEK(734)
```

The result displayed on the screen should be 56.

The syntax for the command is:

```
SYS (51044) { SCREEN MEMORY START} ,{ COLOUR MEMORY START}
```

In using the command, you should bear in mind the limitations noted at the end of the previous section for the position of the screen and colour memory.

HSCREEN *must* be used before calling up the high resolution screen, unless some other method of defining the screen location is used.

4. CLEARING THE HIGH RESOLUTION SCREEN (HCLEAR)

The object of the routine which follows is to perform a screen clear in high resolution mode, setting the value of all the screen memory locations to zero and resetting the colour memory to current foreground and background colours. You can globally set the foreground and background colours using HCLEAR by:

1) Foreground: Printing the appropriate colour control character then calling HCLEAR.
2) Background: POKEing the relevant VIC register at 53281 with the value for the specified colour and then calling HCLEAR.

The procedure for HCLEAR is:

1) Find the start of the high resolution screen memory.
2) Clear the screen memory by loading zero into every location.
3) Obtain the current foreground and background colours.
4) Clear the colour memory by loading into each byte the value which will produce the current foreground and background colours.
5) Home the high resolution cursor.
6) Home all low resolution cursor variables.

HClear — BASIC listing

This listing is somewhat more simple than that for the machine code version, in that all the BASIC versions of the high resolution commands so

far have worked on the assumption that the high resolution screen will be placed at 8192, thus removing the need for the special screen parameters routine used by the machine code versions.

```
26000 REM*******************************
26001 REM CLEAR HI-RES
26002 REM*******************************
26010 CC=PEEK(646)
26020 BC=PEEK(53281) AND 15
26030 NC=16*CC+BC
26035 SS=8192 : CM=1024
26040 FOR I=0 TO 8192
26050 POKE SS+I,0
26060 NEXT I
26100 FOR I=0 TO 1023
26110 POKE CM+I,NC
26120 NEXT I
26150 HH=0 : HV=0
26200 RETURN
```

Commentary

Lines 26010–26030: CC will represent the current low resolution foreground colour (ie the colour of the characters) and BC the background colour. These values, which can of course be altered, are what the high resolution screen will work on.

Lines 26035–26120: The two variables are where the routine will expect to find the start of the high resolution screen and the colour memory. The two loops clear the 8K of screen memory and the 1K of colour memory.

HClear — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C79B
C79B               30 SYM
C79B               70 TEMP   = $FD
C79B               80 COLOUR = $0286
C79B               90 VIC    = $D000
C79B              100 HBIT   = $02DF
C79B              110 HLINE  = $02DC
C79B              150 FILL
C79B   84FE       160 STY TEMP+1
C79D   A000       170 LDY #0
C79F   84FD       180 STY TEMP
C7A1              190 L000
C7A1   91FD       200 STA (TEMP),Y
C7A3   C8         210 INY
C7A4   D0FB       220 BNE L000
```

```
C7A6  E6FE      230 INC TEMP+1
C7A8  CA        240 DEX
C7A9  D0F6      250 BNE L000
C7AB  60        260 RTS
C7AC            290 HCLEAR
C7AC  AD18D0    310 LDA VIC+$18
C7AF  0A        320 ASL A
C7B0  0A        330 ASL A
C7B1  2938      340 AND #$38
C7B3  85FD      350 STA TEMP
C7B5  AD8802    360 LDA 648
C7B8  29C0      370 AND #$C0
C7BA  05FD      380 ORA TEMP
C7BC  A8        390 TAY
C7BD  A220      410 LDX #32
C7BF  A900      420 LDA #00
C7C1  209BC7    430 JSR FILL
C7C4  AC8802    450 LDY 648
C7C7  A204      470 LDX #4
C7C9  AD8602    490 LDA COLOUR
C7CC  0A        500 ASL A
C7CD  0A        510 ASL A
C7CE  0A        520 ASL A
C7CF  0A        530 ASL A
C7D0  85FD      540 STA TEMP
C7D2  AD21D0    560 LDA VIC+$21
C7D5  290F      570 AND #$0F
C7D7  05FD      580 ORA TEMP
C7D9  209BC7    600 JSR FILL
C7DC            620 HHOME
C7DC  A900      630 LDA #0
C7DE  8DDF02    640 STA HBIT
C7E1  8DDC02    650 STA HLINE
C7E4  4C66E5    670 JMP $E566
C7E7            680 END
  TOTAL ERRORS IN FILE --- 0

TEMP            FD
COLOUR          286
VIC             D000
HBIT            2DF
HLINE           2DC
FILL            C79B
L000            C7A1
HCLEAR          C7AC
HHOME           C7DC
  TOTAL NUMBER OF SYMBOLS --- 9
```

Machine code

```
ADD   DATA                          CHECKSUM

C79B  84 FE A0 00 84 FD 91 FD   9FB3
```

```
C7A3  C8 D0 FB E6 FE CA D0 F6   D36E
C7AB  60 AD 18 D0 0A 0A 29 38   6C42
C7B3  85 FD AD 88 02 29 C0 05   A219
C7BB  FD A8 A2 20 A9 00 20 9B   C4E3
C7C3  C7 AC 88 02 A2 04 AD 86   A6A0
C7CB  02 0A 0A 0A 0A 85 FD AD   0A6B
C7D3  21 D0 29 0F 05 FD 20 9B   4F87
C7DB  C7 A9 00 8D DF 02 8D DC   9F86
C7E3  02 4C 66 E5               02F1
```

Commentary

Lines 150–260: This is a subroutine, called by the main part of HCLEAR, to do the work of loading whatever byte is held in the screen memory into a specified area of RAM, which for our purposes will be either the high resolution screen memory or the colour memory. The start page (256 bytes) of the block will already have been stored in the Y register and the address which this represents is transferred to the two bytes at TEMP, with the least significant byte set to zero. Indexed addressing is used to store the contents of register A (which will be zero if it is screen memory which is being cleared) in successive locations as indicated by the address in the two bytes at TEMP plus the value in the Y register. The main part of the routine has stored the number of pages to be cleared (32 for the screen and four for the colour memory) in the X register and every time 256 bytes have been dealt with, the address in TEMP is increased by one page and X decremented by one. When X reaches zero, the process is complete.

Lines 310–390: These lines obtain from [SCREEN POINTERS] the position of the screen memory in the current video block and combine it with the value of the current video block as obtained from the two most significant bytes of the register at 648 to provide the start address in the Y register for the clearing of the screen.

Lines 410–430: The screen consists of 32 pages of 256 bytes each, so 32 is loaded into the X register. The value to be placed into each screen byte is zero, and this is placed in the accumulator.

Line 450: The process of obtaining the start of the colour memory is relatively simple because we have already altered the pointer at 648 to indicate the start of colour memory when entering high resolution mode.

Line 470: Colour memory is 1K long and so consists of four 256-byte pages.

Lines 490–600: The value of the current character-printing colour is

obtained from the register at $286 and transformed by left shifts into the four most significant bits in the accumulator. This value is then stored for a moment in TEMP. The current background colour is obtained from the VIC register at $D021 (53281), with all but the four least significant bits being stripped away by ANDing with 15. The resulting value is combined with that in TEMP to provide a value in the accumulator where the four high bits indicate the foreground colour and the four low bits the background. This is the arrangement used by the colour memory. Once this value has been placed in the accumulator, FILL can be called to clear the colour memory.

Lines 620–670: As explained in the introduction to this section, HCLEAR also homes the 'cursor'. Quite what this means will be explained in the next section on HLOCATE. For the moment, you might like to note that these lines set two registers used by the high resolution cursor routines to define horizontal and vertical position and also call upon a ROM routine which automatically resets various other cursor variables.

Testing

BASIC: Enter the routine along with the high resolution on/off routines and add two new lines as follows:

```
22600 GOSUB 26000
22900 GET T$ : IF T$ = "" THEN 22900
```

Now RUN, and this time, instead of seeing the garbage screen appear and disappear, you will see it appear and then, when a key is pressed, begin to clear, leaving a screen of character-sized coloured blocks. These are the high resolution screen interpreting the letters on the low-resolution screen, since that is what is now being used as colour memory. Finally the coloured blocks themselves will clear, leaving the screen the same colour as it was under low resolution and the system will then return to low resolution.

Note that, if you use the routine twice without switching off the machine, you will find that the screen enters the coloured block stage immediately and appears to be locked up. The simple reason for this is that the previous use of the routine has already cleared the screen. HCLEAR is proceeding as usual but, since there is nothing to be cleared, it appears to have stopped.

You might like to try adding these lines, which have the effect of moving the colour memory away from the location of the normal low resolution screen to the 1K block beginning at 8192:

```
26035 SS = 8192 : CM = 8192
26037 POKE 53272, CM/64 OR 8
```

RUN the program again and you will find that, on returning to BASIC, the

low resolution screen has not been cleared during the high resolution phase. You will also see that, on the second part of HCLEAR, when the colour is being set up, the top of the high resolution screen appears to be being corrupted. This is a graphic illustration of the use of the high resolution colour memory. What we have done here is to move the high resolution colour memory so that it occupies the first 1K of what is also the screen memory. When the colour value is placed into the colour memory, the data also appears in the top one-eighth of the screen itself.

Colour memory cannot usefully be located anywhere other than at 1024 when occupying block one because of the shadow character data which stretched from 4096 to 8191 and the BASIC program starting at 2048. Changing video bank or shifting the start of BASIC up by 1K will make room for a colour memory which does not interfere with anything else.

MACHINE CODE: The routine may be tested by the use of the BASIC high resolution test program given before. Add the following three lines to the short BASIC program used for testing the machine code versions of HIGH and LOW:

```
 3 POKE 53281,(PEEK (53281) + 1) AND 15
25 SYS 51116
27 FOR I = 1 TO 3000 : NEXT
```

When this is run, the screen will flash another colour, then enter high resolution mode. The indication that the screen has been cleared is that the blank high resolution screen will be set to a new colour. After a pause, the screen should return to low resolution mode. This test can be RUN time and time again because line 3 will always lead to a change of the current background colour and so to a change of the high resolution screen colour. Without such a change, it is difficult to tell that the screen is being cleared after the first time.

Syntax and use

The syntax for HCLEAR is:

```
SYS 51116
```

The command should only be used when the high resolution screen memory has been properly set aside and the screen is *on*. There is no built-in protection against using HCLEAR in low resolution mode. The reason for this is that HCLEAR will normally be protected by the fact that it will not be called directly. Once you have entered all the routines from the high resolution section of this book you will find that HCLEAR will be called by printing CLS in the high resolution mode. In low resolution mode the CLS character will work normally and HCLEAR will not be called.

5. LOCATING THE HIGH RESOLUTION CURSOR (HLOCATE)

The purpose of this routine is to allow the high resolution cursor position to be moved at will in much the same way as the low resolution cursor position routine given earlier.

To understand the working of the high resolution cursor, it is necessary to recap a little on the functioning of the cursor in low resolution. Two pointers in zero-page memory are used by the system to control the position of the cursor. A two-byte pointer at $D1 holds the start address in memory of the screen line on which the cursor is currently positioned. In addition a single byte at $D3 records the position of the cursor along that line.

In order to position a high resolution cursor, we need not only the information as to the character position which is contained within addresses $D–3, but also two more pointers to the correct byte and bit.

For the machine code version of the command, these two pointers, both of which will hold values in the range of zero to seven, will be placed in two bytes unused by BASIC, namely $02DF and $02DC for byte and bit respectively. The BASIC version will hold the values in two simple variables, C1 and C2.

The procedure for HLOCATE is as follows:

1) Obtain the high resolution X and Y coordinates.
2) Derive from these the character position on the screen in which the particular pixel will fall.
3) Derive from the vertical coordinate, the byte within the character position.
4) Derive from the horizontal element the bit position within the byte.
5) Set the low resolution cursor using the kernal routine.
6) Set the values of the variables which store the byte and bit values for the specified pixel.

Note that the routine does not *do* anything visible, it is a utility which will be called upon by later routines.

HLocate — BASIC listing

```
27000 REM*******************************
27001 REM SET HIGH RESOLUTION CURSOR
27002 REM*******************************
27010 HH=160 : HV=100
27020 LH=INT(HH/8) : LV=INT(HV/8)
27030 POKE 781,LV
27040 POKE 782,LH
27050 SYS (65520)
27100 C1=SS + 320*LV + 8*LH + (HV AND 7)
27110 C2=7-(HH AND 7)
27200 RETURN
```

Commentary

Line 27010: A temporary line providing values for the horizontal and vertical coordinates, expressed in terms of the high resolution screen. The position specified is in the middle of the screen.

Line 27020: The high resolution values are translated into their low resolution character position. The pixel in question will be at 20,12 expressed in terms of character squares.

Lines 27030–27050: A call to the kernal routine which sets the position of the cursor. The same method was used in LOCATE.

Line 27100: This line derives the byte which will hold the pixel position on the screen, using the method of calculation described in the introduction to this chapter.

Line 27110: This line looks a little confusing at first but is quite simple, its purpose being to calculate the *bit* within byte C1, again using the method described at the beginning of the chapter.

HLocate — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C7E7
C7E7               30 SYM
C7E7               90 LOLINE = $00D1
C7E7              100 LOCOL  = $00D3
C7E7              110 CURSOR = $00F3
C7E7              120 HILINE = $02DF
C7E7              130 HIBIT  = $02DC
C7E7              140 GETBYT = $C006
C7E7              150 GETWRD = $C000
C7E7              160 VIC    = $D000
C7E7              170 COMMA  = $AEFD
C7E7              200 GETCUR
C7E7   18         210 CLC
C7E8   A5D1       220 LDA LOLINE
C7EA   65D3       230 ADC LOCOL
C7EC   85F3       240 STA CURSOR
C7EE   A5D2       250 LDA LOLINE+1
C7F0   2903       260 AND #$3
C7F2   6900       270 ADC #0
C7F4   06F3       290 ASL CURSOR
C7F6   2A         300 ROL A
C7F7   06F3       310 ASL CURSOR
C7F9   2A         320 ROL A
C7FA   06F3       330 ASL CURSOR
```

```
C7FC  2A        340 ROL A
C7FD  85F4      350 STA CURSOR+1
C7FF  AD18D0    370 LDA VIC+$18
C802  290E      380 AND #$0E
C804  0A        390 ASL A
C805  0A        400 ASL A
C806  18        410 CLC
C807  65F4      420 ADC CURSOR+1
C809  293F      440 AND #$3F
C80B  85F4      450 STA CURSOR+1
C80D  AD8802    470 LDA 648
C810  29C0      480 AND #$C0
C812  05F4      490 ORA CURSOR+1
C814  85F4      500 STA CURSOR+1
C816  ADDF02    520 LDA HILINE
C819  05F3      530 ORA CURSOR
C81B  85F3      540 STA CURSOR
C81D  60        550 RTS
C81E            580 HIAT
C81E  2000C0    600 JSR GETWRD
C821  A003      620 LDY #3
C823  A900      630 LDA #0
C825            640 L000
C825  4615      650 LSR $15
C827  6614      660 ROR $14
C829  6A        670 ROR A
C82A  88        680 DEY
C82B  D0F8      690 BNE L000
C82D  4A        700 LSR A
C82E  4A        710 LSR A
C82F  4A        720 LSR A
C830  4A        730 LSR A
C831  4A        740 LSR A
C832  48        750 PHA
C833  A515      770 LDA $15
C835  D02E      780 BNE IQERR
C837  A514      790 LDA $14
C839  C928      800 CMP #40
C83B  B028      810 BCS IQERR
C83D  48        820 PHA
C83E  20FDAE    830 JSR COMMA
C841  2006C0    850 JSR GETBYT
C844  C9C8      860 CMP #200
C846  B01D      870 BCS IQERR
C848  A003      890 LDY #3
C84A  A900      900 LDA #0
C84C            910 L001
C84C  4614      920 LSR $14
C84E  6A        930 ROR A
C84F  88        940 DEY
C850  D0FA      950 BNE L001
```

```
C852  4A          960 LSR A
C853  4A          970 LSR A
C854  4A          980 LSR A
C855  4A          990 LSR A
C856  4A         1000 LSR A
C857  8DDF02     1010 STA HILINE
C85A  68         1030 PLA
C85B  A8         1040 TAY
C85C  68         1050 PLA
C85D  8DDC02     1060 STA HIBIT
C860  A614       1070 LDX $14
C862  4CF0FF     1080 JMP $FFF0
C865             1090 IQERR
C865  4C48B2     1100 JMP $B248
C868             1110 END

 TOTAL ERRORS IN FILE --- 0

LOLINE           D1
LOCOL            D3
CURSOR           F3
HILINE           2DF
HIBIT            2DC
GETBYT           C006
GETWRD           C000
VIC              D000
COMMA            AEFD
GETCUR           C7E7
HIAT             C81E
L000             C825
L001             C84C
IQERR            C865
 TOTAL NUMBER OF SYMBOLS --- 14
```

Machine code

```
ADD   DATA                                  CHECKSUM

C7E7  18 A5 D1 65 D3 85 F3 A5                 60E7
C7EF  D2 29 03 69 00 06 F3 2A                 7C58
C7F7  06 F3 2A 06 F3 2A 85 F4                 4F9E
C7FF  AD 18 D0 29 0E 0A 0A 18                 79D4
C807  65 F4 29 3F 85 F4 AD 88                 826A
C80F  02 29 C0 05 F4 85 F4 AD                 2FD9
C817  DF 02 05 F3 85 F3 60 20                 88A4
C81F  00 C0 A0 03 A9 00 46 15                 4A19
C827  66 14 6A 88 D0 F8 4A 4A                 58FE
C82F  4A 4A 4A 48 A5 15 D0 2E                 4C8A
C837  A5 14 C9 28 B0 28 48 20                 79F0
C83F  FD AE 20 06 C0 C9 C8 B0                 89C4
C847  1D A0 03 A9 00 46 14 6A                 431A
C84F  88 D0 FA 4A 4A 4A 4A 4A                 A036
```

```
C857   8D DF 02 68 A8 68 8D DC    8DD6
C85F   02 A6 14 4C F0 FF 4C 48    3E1C
C867   B2                        00B2
```

Commentary

Lines 200–550: These lines constitute a subroutine to determine the correct byte within the memory to correspond with the specified position for the cursor on the high resolution screen.

Lines 200–270: The address of the low resolution cursor is obtained from the low resolution cursor pointers. The value for the column on the screen is added to the least significant byte of the line start address and this is stored in CURSOR. The most significant byte of the line start address is ANDed with 3, in order to remove any reference to any value above 1K. What this means is that, in CURSOR (least significant byte) and the accumulator (most significant byte), we no longer have a pointer to an address but to a position within 1K screen.

Lines 290–350: The combined value in CURSOR and the accumulator is now shifted left three times, multiplying it by eight. The position within a hypothetical 1K screen has now been translated into the start position of a character square in a screen with eight bytes per character square.

Lines 370–420: These lines extract the address of the start of the screen within the current video bank and add to it the position of the byte within the screen.

Lines 440–450: ANDing the most significant byte with $3F ensures that the two most significant bits are cleared, ie the bits which will eventually determine the 16K block.

Lines 470–500: The two most significant bits of the pointer held at 648 in memory are used to add in the start position of the current video block. We now have an address in memory to correspond to the first byte of the correct character position on the high resolution screen.

Lines 520–550: The line within the character position, which has previously been calculated by the main routine, is now added to the address of the start byte. We have now specified the address of the byte to be addressed.

Lines 600–750: The HLOCATE command will have two parameters, X for horizontal position on the screen, and Y for vertical position. GETWRD is used to pick up the X parameter. The horizontal position on

the screen is now divided by eight, thus obtaining the character position across the screen. The sequence of rotations to the right which are carried out on the value obtained by GETWRD in $14-15 accomplishes this division and leaves the three least significant bits of that value shifted into the accumulator as the three *most* significant bits. Five left shifts move these bits down to positions 0–2 in the accumulator. These three bits now represent that part of the horizontal component which cannot be expressed in bytes, ie the position of a bit within the horizontal byte. This bit position value is pushed temporarily onto the stack.

Lines 770–830: The value in $14–15, which represents now the horizontal byte position, stripped of any reference to the bit within that byte, is tested to see that it is not outside the range 0–39. Clearly it fails this test if there is anything at all in the most significant byte. The least significant byte needs only to be tested against 40, with an error being generated if it equals or exceeds the limit. The byte component of the horizontal component is finally pushed on to the stack.

Lines 850–870: The vertical or Y component of the position is now picked up by GETBYT and checked against the maximum value of 24.

Lines 880–1000: The vertical component, or high resolution line number, is treated by these lines in the same way as the horizontal component was in lines 620–740. The value obtained is divided by eight, in order to obtain the character position down the screen, with the remainder indicating the particular byte in that character position.

Line 1010: The pointer used for the vertical component is updated.

Lines 1030–1070: The pointer used to store the bit value of the horizontal component is updated. Registers X and Y are loaded with the line and column values.

Line 1080: This jump is to the low resolution routine in the kernal which plots the position of the cursor. Note that, to call this routine, the carry flag needs to be clear and this was accomplished by the LSR instruction in line 1000.

Testing

BASIC: The BASIC routine is easier to test than the machine code, for the simple reason that we can add a test line to show that the low resolution cursor *has* moved and that the values of the byte and bit are sensible.

Add the following lines to the collection of high resolution routines:

```
22700 GOSUB 27000
27060 PRINT "HERE"
27120 PRINT C1,C2
```

RUN the whole program and, when the screen returns to low resolution, you should find that the word 'here' is printed in the middle of the screen and that the next line has 12196 and 6 as the values for the byte and bit.

MACHINE CODE: It is not possible to test the full high resolution effects of this routine until the following sections have been entered. Fortunately, the routine also has effects in low resolution mode:

1) It is capable of moving the low resolution cursor around the screen.
2) It places the correct values into the storage locations which will be used by the later high resolution routines.

In order to test the routine, therefore, in normal low resolution mode enter:

```
SYS (51230) 0,0 : PRINT "*"
```

This should home the cursor and print an asterisk in the top lefthand corner of the screen. PEEKing the contents of HIBIT and HILINE should result in two zeros. Now enter:

```
SYS (51230) 256,0 : PRINT "*"
```

The asterisk should now appear in position 32 of the top line, again with the contents of HIBIT and HILINE zero.

Syntax and use

The syntax for the command is:

```
SYS (51230) { COLUMN} ,{ ROW}
```

where COLUMN is in the range 0–319 and ROW is in the range 0–199, with both components being numbered from zero in the top lefthand corner of the screen. If either of the parameters fall outside the correct range, then an ILLEGAL QUANTITY error message will be generated.

6. PLOTTING PIXELS IN HIGH RESOLUTION (HPLOT)

This routine is similar in purpose to the low resolution PLOT routine given previously, except that the purpose of the present routine is to plot, unplot or invert a single pixel on the high resolution screen according to the mode specified.

The procedure for HPLOT is:

1) Obtain the MODE parameter.
2) Obtain the coordinates of the pixel to be plotted.

3) Position the low and high resolution cursor with the HLOCATE routine.
4) Perform the set, reset or inversion of the pixel specified.
5) Ensure that the colour for the character square in which the pixel is to be plotted is the current character colour.

HPlot — BASIC listing

```
28000 REM*******************************
28001 REM HIGH RESOLUTION PLOTTING
28002 REM*******************************
28010 FOR MODE = 0 TO 2
28050 FOR HH=100 TO 200+50*(MODE=1)
28060 HV=100 : GOSUB 27000
28200 MASK=2^C2 : BYTE=PEEK(C1)
28210 IF MODE=0 THEN BYTE=BYTE OR MASK
28220 IF MODE=1 THEN BYTE=BYTE AND (255-
MASK)
28230 IF MO=2 THEN BYTE=BYTE+MASK+2*MASK
*((BYTE AND MASK)=MASK)
28240 POKE C1,BYTE
28300 GOSUB 28500
28400 NEXT HH
28410 NEXT MODE
28450 RETURN
28500 REM*******************************
28501 REM SET CHARACTER COLOUR
28502 REM*******************************
28510 CC=PEEK(646)
28520 LP=1024 + 40*LV + LH
28530 POKE LP,(PEEK (LP) AND 15) OR 16*C
C
28550 RETURN
```

Commentary

The loops given are merely a means to set the values of MODE and HH so that their effect can be seen. Before this routine will run, you will need to delete line 27010 from the previous BASIC routine and also any temporary lines you included during testing.

Line 28010: MODEs are as in the low resolution plotting routine, with zero for plotting, one for unplotting and two for inverting.

Line 28050: This loop will move the value of HH along the horizontal axis from 100 to 200 when MODE is either one or two, but only from 100 to 150 when MODE is one.

Line 28060: The vertical coordinate will always be 100 so a straight line should be plotted.

Lines 28200–28240: The creation and application of a mask to the byte C1, depending upon whether bit C2 is to be set, reset or inverted.

Line 28300: With this GOSUB, the system register which stores the printing colour is PEEKed and the result placed into the upper four bits of the colour memory location corresponding to the current character square. Any pixels in that square will be changed to the current printing colour.

HPlot — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C868
C868               30 SYM
C868              120 GETBYT = $C006
C868              130 IQERR  = $C865
C868              140 COMMA  = $AEFD
C868              150 GETCUR = $C7E7
C868              160 HIBIT  = $02DC
C868              170 HILINE = $02DF
C868              180 CURSOR = $00F3
C868              190 HIAT   = $C81E
C868              200 LOLINE = $00D1
C868              210 LOCOL  = $00D3
C868              220 COLOUR = $0286
C868              250 HPLOT
C868  2006C0      270 JSR GETBYT
C86B  C903        280 CMP #3
C86D  B0F6        290 BCS IQERR
C86F  48          300 PHA
C870  20FDAE      310 JSR COMMA
C873  201EC8      330 JSR HIAT
C876  20E7C7      340 JSR GETCUR
C879  68          350 PLA
C87A  AA          360 TAX
C87B  38          380 SEC
C87C  A900        390 LDA #0
C87E  ACDC02      400 LDY HIBIT
C881              410 L000
C881  6A          420 ROR A
C882  88          430 DEY
C883  10FC        440 BPL L000
C885  C8          460 INY
C886  CA          480 DEX
C887  3008        490 BMI SET
C889  CA          500 DEX
C88A  300A        510 BMI CLEAR
C88C  51F3        530 EOR (CURSOR).Y
C88E  4C9AC8      540 JMP L001
C891              560 SET
C891  11F3        570 ORA (CURSOR).Y
```

```
C893  4C9AC8     580 JMP L001
C896              600 CLEAR
C896  49FF        610 EOR #$FF
C898  31F3        620 AND (CURSOR).Y
C89A              630 L001
C89A  91F3        640 STA (CURSOR).Y
C89C  A4D3        660 LDY LOCOL
C89E  B1D1        680 LDA (LOLINE).Y
C8A0  290F        690 AND #$F
C8A2  91D1        700 STA (LOLINE).Y
C8A4  AD8602      720 LDA COLOUR
C8A7  0A          730 ASL A
C8A8  0A          740 ASL A
C8A9  0A          750 ASL A
C8AA  0A          760 ASL A
C8AB  11D1        770 ORA (LOLINE).Y
C8AD  91D1        780 STA (LOLINE).Y
C8AF  60          790 RTS
C8B0              800 END
 TOTAL ERRORS IN FILE --- 0


GETBYT            C006
IQERR             C865
COMMA             AEFD
GETCUR            C7E7
HIBIT             2DC
HILINE            2DF
CURSOR            F3
HIAT              C81E
LOLINE            D1
LOCOL             D3
COLOUR            286
HPLOT             C868
L000              C881
SET               C891
CLEAR             C896
L001              C89A
 TOTAL NUMBER OF SYMBOLS --- 16
```

Machine code

```
ADD   DATA                        CHECKSUM

C868  20 06 C0 C9 03 B0 F6 48     3B1C
C870  20 FD AE 20 1E C8 20 E7     6C37
C878  C7 68 AA 38 A9 00 AC DC     9DBC
C880  02 6A 88 10 FC C8 CA 30     3A44
C888  08 CA 30 0A 51 F3 4C 9A     44A6
C890  C8 11 F3 4C 9A C8 49 FF     94E1
C898  31 F3 91 F3 A4 D3 B1 D1     812F
C8A0  29 0F 91 D1 AD 86 02 0A     3EFE
```

```
C8A8  0A 0A 0A 11 D1 91 D1 60    149E
C8B0                             0000
```

Commentary

Lines 260–310: These lines pick up the mode parameter and check that it is in the range 0–2, otherwise an ILLEGAL QUANTITY error message will be printed. The mode value is temporarily stored on the stack.

Lines 330–340: The two previous routines HLOCATE and GETCUR are used to position the cursor correctly.

Lines 350–360: The mode value is pulled back from the stack and placed into the X register.

Lines 380–440: After calling HLOCATE and GETCUR the various high resolution pointers have been set up. One of these is HIBIT, which contains a value in the range 0–7 which indicates which bit of the relevant byte controls the pixel specified. These lines create a mask. The lines first of all set the carry flag, and then load the accumulator with zero. The number of the bit, as recorded in HIBIT, is loaded into the Y register. A series of rotations to the right are performed on the accumulator, according to the value stored in the Y register. With the first rotation, the '1' in the carry flag is moved into the position of the most significant bit (bit seven); while the second rotation (if there is one) will move the '1' to bit six. The BPL instruction ensures that there will be one more rotation than the value stored in Y.

An apparent illogicality needs to be noted here about the direction in which the accumulator is rotated. If, for example, the HLOCATE routine had determined that bit zero in the relevant screen memory byte was being referred to, these lines would result in zero being placed into the Y register and a single rotation being made. This would place a set bit, not in position zero but in position seven. This is not an error; it is simply that, when it comes to the plotting of the screen the HLOCATE routine has counted the bit position from the righthand end of the byte, the end normally called bit zero. Where screen memory is concerned, however, bits are regarded as going from left to right.

Line 460: The Y register, which was left at −1, or 255, by the loop in the previous lines, is set to zero.

Lines 480–510: The mode value has previously been stored in X. These lines examine the value in the X register to see which mode is called for.

Lines 530–540: If the mode is two, the mask is used to perform an XOR on the relevant screen byte. Since the mask contains only one set bit, that referring to the pixel specified, this means that, if that bit is already set in

the screen byte, then the pixel will be turned off by resetting the bit. If the relevant bit in the screen byte is not set, then the XOR will set it and thus turn the pixel on. From this we can see that mode two is, in fact, the invert pixel mode.

Lines 560–580: The mode for switching on a pixel is zero and the process is performed by these lines.

Lines 600–630: The final mode, one, is to clear a pixel. This is done by inverting the mask. The mask with its one set bit is XORed with 255. The result is a mask in which every bit is set *except* the one which was set originally in the mask. All that needs to be done now is to AND the new mask with the relevant screen byte.

Line 640: The value resulting from the set, clear or invert process is now placed on to the screen.

Lines 660–690: Whenever a pixel is plotted, unplotted or inverted, the character position in which it falls is set to the current printing colour. In preparation for this, the upper four bits of the relevant byte of colour memory are cleared by ANDing them with $F (15).

Lines 720–780: The current printing colour is stored by the 64 at $0286. This value is picked up and stored into the upper four bits of the relevant byte of colour memory.

Testing

BASIC: The BASIC routine is self-testing. First delete line 27010, then change 22700 to read:

```
22700 GOSUB 28000
```

When RUN it should first draw a line (MODE 0), then erase half of it (MODE 1) and finally redraw the first half and erase the second (MODE 2). If it works correctly, it can be incorporated into your own programming by removing the loops and using other means to set the values of HH and HV.

MACHINE CODE: To test the routine, take the BASIC program used to test HIGH, LOW and HCLEAR, delete lines 3 and 30 then add the following:

```
40 FOR MODE = 0 TO 2
45 FOR I = 100 TO 200 + 50*(MODE = 1)
50 SYS (51304) MODE,I,100
```

```
60 NEXT I, MODE
65 FOR I = 1 TO 1000
70 SYS 50944
```

What should happen is the same as with the BASIC routine except, of course, faster.

Syntax

Syntax for HPLOT is:

SYS (51304) { MODE} ,{ COLUMN} ,{ ROW}

where ROW and COLUMN are as in the HLOCATE routine, and MODE is as follows:

0 — set specified pixel
1 — clear specified pixel
2 — invert specified pixel

7. HIGH RESOLUTION LINE DRAWING

Part 1. Some simple arithmetic (INTEGER)

The purpose of the following sections is to draw the best possible approximation of a straight line between two points specified by the user. Compared to a similar routine in BASIC, the machine code which will be required is complex. The reason for this is that line drawing requires a number of routines to perform arithmetic operations on 16-bit numbers and these have to be supplied in full since the arithmetic routines in the ROM are unsuited to the purpose. In addition the 6510 chip cannot deal directly with 16-bit arithmetic, or rather '15-bit signed arithmetic', where one bit is employed to record whether the value in question is positive or negative. In this section, you will find the simple arithmetic which will be used in what follows.

No attempt is made to simulate the routines in BASIC.

Integer — assembly language listing

```
ADD.   DATA      SOURCE CODE
00               10 PRT
00               20 ORG $C8B0
C8B0             30 SYM
C8B0             70 ABS
C8B0   B501      90 LDA 1.X
C8B2   100D     100 BPL L000
C8B4            120 NEGATE
C8B4   38       130 SEC
```

```
C8B5  A900      140 LDA #0
C8B7  F500      150 SBC 0.X
C8B9  9500      160 STA 0.X
C8BB  A900      170 LDA #0
C8BD  F501      180 SBC 1.X
C8BF  9501      190 STA 1.X
C8C1            200 L000
C8C1  60        210 RTS
C8C2            240 ADD
C8C2  18        250 CLC
C8C3  B500      260 LDA 0.X
C8C5  790000    270 ADC 0.Y
C8C8  9500      280 STA 0.X
C8CA  B501      290 LDA 1.X
C8CC  790100    300 ADC 1.Y
C8CF  9501      310 STA 1.X
C8D1  60        320 RTS
C8D2            350 MINUS
C8D2  38        360 SEC
C8D3  B500      370 LDA 0.X
C8D5  F90000    380 SBC 0.Y
C8D8  9500      390 STA 0.X
C8DA  B501      400 LDA 1.X
C8DC  F90100    410 SBC 1.Y
C8DF  9501      420 STA 1.X
C8E1  60        430 RTS
C8E2            460 MOVE
C8E2  B90000    470 LDA 0.Y
C8E5  9500      480 STA 0.X
C8E7  B90100    490 LDA 1.Y
C8EA  9501      500 STA 1.X
C8EC  60        510 RTS
C8ED            540 SWAP
C8ED  E8        550 INX
C8EE  C8        560 INY
C8EF  20F4C8    570 JSR SWAP1
C8F2  CA        580 DEX
C8F3  88        590 DEY
C8F4            610 SWAP1
C8F4  B500      620 LDA 0.X
C8F6  48        630 PHA
C8F7  B90000    640 LDA 0.Y
C8FA  9500      650 STA 0.X
C8FC  68        660 PLA
C8FD  990000    670 STA 0.Y
C900  60        680 RTS
C901            710 TEST
C901  B501      720 LDA 1.X
C903  D90100    730 CMP 1.Y
C906  D005      740 BNE L001
C908  B500      750 LDA 0.X
```

```
C90A  D90000      760 CMP 0.Y
C90D              770 L001
C90D  60          780 RTS
C90E              810 TIMES2
C90E  1600        830 ASL 0.X
C910  3601        840 ROL 1.X
C912  B501        860 LDA 1.X
C914  297F        870 AND #$7F
C916  9002        880 BCC L002
C918  0980        890 ORA #$80
C91A              900 L002
C91A  9501        910 STA 1.X
C91C  60          920 RTS
C91D              930 END
 TOTAL ERRORS IN FILE --- 0

ABS               C8B0
NEGATE            C8B4
L000              C8C1
ADD               C8C2
MINUS             C8D2
MOVE              C8E2
SWAP              C8ED
SWAP1             C8F4
TEST              C901
L001              C90D
TIMES2            C90E
L002              C91A
 TOTAL NUMBER OF SYMBOLS --- 12
```

Machine code

```
ADD   DATA                          CHECKSUM
C8B0  B5 01 10 0D 38 A9 00 F5        62E9
C8B8  00 95 00 A9 00 F5 01 95        343B
C8C0  01 60 18 B5 00 79 00 00        28B4
C8C8  95 00 B5 01 79 01 00 95        6591
C8D0  01 60 38 B5 00 F9 00 00        2EB4
C8D8  95 00 B5 01 F9 01 00 95        6991
C8E0  01 60 B9 00 00 95 00 B9        32AD
C8E8  01 00 95 01 60 E8 C8 20        1B80
C8F0  F4 C8 CA 88 B5 00 48 B9        D4B1
C8F8  00 00 95 00 68 99 00 00        1844
C900  60 B5 01 D9 01 00 D0 05        6C9D
C908  B5 00 D9 00 00 60 16 00        774C
C910  36 01 B5 01 29 7F 90 02        3656
C918  09 80 95 01 60                 0746
```

Commentary

Lines 90–100: These lines check to see whether a number pointed to by the

X register is positive or negative. If it is already positive then the following section of code is jumped over.

Lines 120–210: These lines are only activated from the previous lines if the value being worked upon is negative, though they are called from other places where this will not apply. Their function at this point is to extract the absolute value of the negative number. In general, however, what they do is transform the sign of any number. The procedure for doing this is to subtract the number in question from zero, for instance:

$0-(-1)=1$

Both bytes of the negative 15-bit value are subtracted from the contents of the accumulator, which has previously been loaded with zero.

Lines 240–320: These lines add two 15-bit signed numbers pointed to by the X and Y registers and store the result in the place originally occupied by the value pointed to by the X register.

Lines 350–430: Similar to the previous lines except that subtraction is carried out, the number indicated by the X register being subtracted from that indicated by the Y register.

Lines 460–510: These lines move a 16-bit number pointed to by the Y register into a location indicated by the X register.

Lines 540–680: The contents of the two-byte addresses pointed to by the X and Y registers are swapped by these lines.

Lines 700–780: These lines carry out a comparison of the two values pointed to by the X and Y registers. The results of the comparison are contained in the zero flag and the carry flag (which is set at the beginning of the section). If the two values are equal, the zero flag is set. If the X value is greater than or equal to the Y value, the carry flag will remain set. If the carry flag is clear, it is an indication that the Y value is greater than the X value.

Lines 800–920: These lines perform a simple multiplication by two on the value pointed to by the X register. The only complication is the necessity to ensure that the sign bit is not lost when it is rotated into the carry flag.

Testing

These routines cannot be effectively tested until the subsequent sections have been entered.

Part 2. Window test

The purpose of this routine is to act as an 'interface' between the line and

circle routines which follow and the HPLOT routine which you have already entered. The line drawing routines provide the coordinates for the plotting of individual points and this routine sets up the variables necessary for HPLOT, testing that the point to be plotted falls within the screen.

Since the routine is intended only to interface with the machine code commands, no attempt has been made to simulate it in BASIC.

Window — assembly language listing

```
ADD.  DATA      SOURCE CODE
00              10 PRT
00              20 ORG $C91D
C91D            30 SYM
C91D            90 TEMP   = $14
C91D           100 AVAR   = $6B
C91D           110 BVAR   = $6F
C91D           120 XHI    = $4B
C91D           130 XLO    = $4D
C91D           140 YHI    = $4F
C91D           150 YLO    = $51
C91D           160 MODE   = $02
C91D           170 TEST   = $C901
C91D           180 MOVE   = $C8E2
C91D           190 DOHIAT = $C848
C91D           200 DOHPLT = $C876
C91D           230 L000
C91D  A003     240 LDY #3
C91F  A900     250 LDA #0
C921           260 L001
C921  466C     270 LSR AVAR+1
C923  666B     280 ROR AVAR
C925  6A       290 ROR A
C926  88       300 DEY
C927  D0F8     310 BNE L001
C929  4A       320 LSR A
C92A  4A       330 LSR A
C92B  4A       340 LSR A
C92C  4A       350 LSR A
C92D  4A       360 LSR A
C92E  48       370 PHA
C92F  A56B     380 LDA AVAR
C931  48       390 PHA
C932  A214     400 LDX #TEMP
C934  A06F     410 LDY #BVAR
C936  20E2C8   420 JSR MOVE
C939  4C48C8   430 JMP DOHIAT
C93C           460 WINDOW
C93C  A26B     470 LDX #AVAR
C93E  A04B     480 LDY #XHI
C940  2001C9   490 JSR TEST
```

```
C943  B020      500 BCS L002
C945  A04D      510 LDY #XLO
C947  2001C9    520 JSR TEST
C94A  9019      530 BCC L002
C94C  A26F      540 LDX #BVAR
C94E  A04F      550 LDY #YHI
C950  2001C9    560 JSR TEST
C953  B010      570 BCS L002
C955  A051      580 LDY #YLO
C957  2001C9    590 JSR TEST
C95A  9009      600 BCC L002
C95C  201DC9    620 JSR L000
C95F  A502      630 LDA MODE
C961  48        640 PHA
C962  4C76C8    650 JMP DOHPLT
C965            660 L002
C965  60        670 RTS
C966            680 END
 TOTAL ERRORS IN FILE --- 0


TEMP            14
AVAR            6B
BVAR            6F
XHI             4B
XLO             4D
YHI             4F
YLO             51
MODE            02
TEST            C901
MOVE            C8E2
DOHIAT          C848
DOHPLT          C876
L000            C91D
L001            C921
WINDOW          C93C
L002            C965
 TOTAL NUMBER OF SYMBOLS --- 16
```

Machine code

```
ADD   DATA                          CHECKSUM

C91D  A0 03 A9 00 46 6C 66 6B   6AF7
C925  6A 88 D0 F8 4A 4A 4A 4A   84D6
C92D  4A 48 A5 6B 48 A2 14 A0   57E0
C935  6F 20 E2 C8 4C 48 C8 A2   6DF2
C93D  6B A0 4B 20 01 C9 B0 20   6D8C
C945  A0 4D 20 01 C9 90 19 A2   70AC
C94D  6F A0 4F 20 01 C9 B0 10   6FFC
C955  A0 51 20 01 C9 90 09 20   710A
C95D  1D C9 A5 02 48 4C 76 C8   5AA4
```

```
C965  60                    0060
```

Commentary

Lines 230–430: These lines set up the variables necessary for the execution of the high resolution plot of a single pixel.

Lines 460–600: These are the lines which test to see if the pixel to be plotted falls within the window designated by the next routine. If not, rather than generate an error, the plot instruction would be ignored.

Lines 620–650: At this point the position must be a valid one so a jump is made to the HPLOT routine, with program execution returning from HPLOT to the line drawing routine.

Testing

This routine is necessary for the execution of what follows but cannot be tested effectively at this stage.

Part 3. The line drawing algorithm (LINE PLOTTING)

There are many methods of calculation, or algorithms, available to aid in the plotting of a straight line between two points. Of these perhaps the most widely known and used in microcomputers is 'Bresenham's algorithm'. The basis of this method is to plot a line which has a tendency to be straight along whichever axis, vertical or horizontal, expresses the greatest distance between the start and finish points for the line. As this line is plotted, however, a record is kept of the amount by which the line deviates from the correct position along the other axis. Whenever the deviation amounts to one unit, or pixel, the next pixel moves not only one position along the longest axis but also one pixel along the subsidiary axis.

Suppose that we wished to draw a line from position 0,0, the top left-hand corner of the screen, to position 10,1, a position one pixel down from the top and 10 pixels to the right. The first thing to do is to identify which axis has the greater movement and it is clearly the one where the line must move from zero to ten. We therefore set two variables, X1 and X2, equal to the start and finish points on this axis. Note that, for the purposes of this line drawing routine, the axis along which there is the greatest movement will *always* be called the X axis, regardless of whether it moves across the screen or up and down.

We now have two values set:

X1 = 0
X2 = 10

The two remaining coordinates are now saved in Y1 and Y2:

Y1 = 0
Y2 = 1

The two ends of line are now defined as X1,Y1 and X2,Y2. We now need two more variables to represent the *difference* between the ends of the line on both axes, and we shall call these DX and DY:

DX = X2 – X1 = 10
DY = Y2 – Y1 = 1

We begin the process of plotting the line by setting one more variable, E, which will be called the 'error term', equal to DX times – 1, so that:

E = – DX = – 10

We can now get down to plotting pixels. The first one to be plotted is at X1,Y1, the beginning of the line. In fact, we take a temporary copy of X1 and Y1 into two more variables X and Y, since later we are going to have to change the values and we do not wish to play around with the record of the start position of the line.

Having plotted the pixel we now add to the error term, E, twice the difference between Y1 and Y2, as recorded in DY. Having done this we test to see whether E is now greater than zero. In this case E started off as – 10 and has had two added, so it has not reached zero. We now add one to the X coordinate but, because E failed the test, do nothing to the Y coordinate.

X is now equal to one and Y is equal to zero and we now plot that pixel. The process is repeated five times in all, with the following pixels being plotted:

0,0
1,0
2,0
3,0
4,0
5,0

At this point, E, which has had two added to it six times, has become greater than zero. This is a sign that we need to move along the Y axis since what E actually records is how far we have deviated from the correct position on the Y axis if we were drawing a true straight line. Y is now increased by one, so that the next pixel to be plotted will be 6,1. E has twice DX subtracted from it and thus returns to – 8.

The remaining pixels are plotted at:

7,1
8,1

9,1
10,1

without E ever exceeding zero. By comparing differences we have arrived at the correct position on both axes. Not only that, by adding *twice* the difference on the Y axis to the error term each time, we have even ensured that the one pixel change in the Y axis takes place in the middle of the line rather than at the end, so that the line looks more natural.

This straightforward method can be adapted to cover all circumstances for straight lines. For instance, the method we have just worked through is designed for lines which move positively along the X axis. For a line whose longest axis moves in a negative direction the simple solution is to swap the ends before drawing it. If the Y axis moves negatively, all that needs to be done is to subtract one, rather than add, every time the error term E exceeds zero.

The only really tricky manouevre is when the longest axis is the vertical one rather than the horizontal. Even so, it is easy enough to record the fact and remember that what is being calculated is not the position across and then the position down but, rather, the other way around. When it comes to actually plotting the position on the screen, it will be necessary only to swap the two coordinates.

The actual routines for line plotting are the most complex you will come across in this book, involving, as they do, a variety of calculations of a fairly complex nature. For that reason we will depart from the normal format of the rest of the book, and in this section enter and test the BASIC version. To speed up the execution of the BASIC routine, it is assumed that you will be using it with the high resolution machine code routines in the memory rather than the BASIC example routines entered so far. You can, if you wish, quite easily alter the BASIC program given here so that it will mesh with the BASIC routines given before. The result, however, will be slow and unlikely to be satisfactory.

Line Plotting — BASIC listing

```
1 GOTO 3
2 SAVE"@0:LINE DRAW",8 : VERIFY "0:LINE
DRAW",8 : STOP
3 REM
10 REM ROUTINE TO DRAW STRAIGHT LINES
20 POKE56,12:CLR
30 COLOUR=3072
40 BITMAP=8192
50 SYS (51044) BITMAP,COLOUR
70 SYS 51003
80 SYS 51116
90 GOSUB 2000
170 SYS 50944
```

```
180 END
1000 REM DRAW STRAIGHT LINE
1010 SWAPED = 0
1020 DX = ABS(X2-X1) : DY = ABS(Y2-Y1)
1030 IF DY>DX THEN SWAPED=-1 : T=X2:X2=Y
2:Y2=T:T=Y1:Y1=X1:X1=T : GOTO1020
1040 X=X1 : Y=Y1
1050 IF X2<X1 THEN X1=X2:X2=X:Y1=Y2:Y2=Y
 : GOTO 1040
1060 YS = SGN(Y2-Y1)
1070 E = -DX
1080 FOR X = X1 TO X2
1090 A = X : B = Y
1100 IF SWAPED THEN B = X : A = Y
1110 IF A<320 AND A>=0 AND B<199 AND B>=
0 THEN SYS (51304) 0,A,B
1120 E = E+2*DY
1130 IF E>0 THEN Y = Y+YS : E = E-2*DX
1140 NEXT
1150 RETURN
2000 REM TEST LINE DRAW
2010 FOR I1 = 10 TO 310 STEP 300
2020 FOR I2 = 10 TO 190 STEP 25
2025 X2 = I1  : Y2 = I2
2030 X1 = 160 : Y1 = 100
2040 GOSUB 1000
2050 NEXT I2,I1
2060 GET T$  : IF T$="" THEN 2060
2070 RETURN
```

Commentary

Lines 20–180: These lines set up and clear the high resolution screen using HSCREEN, HIRES and HCLEAR, then call up the line-drawing test part of the program, finally restoring the screen to low resolution.

Lines 1000–1150: The line-drawing algorithm, which will be commented upon in sections.

Lines 1010–1030: The routine will always work on the assumption that the greatest degree of movement for the line will be along the X axis. This does not mean that the movement will always be across the screen in the way we would normally define the X axis, but simply that the names of the coordinates will be swapped in order to ensure that the difference between X1 and X2 is always greater than or equal to the difference between Y1 and Y2.

Lines 1040–1050: The routine is designed to work with lines which travel in a positive direction in terms of the X axis. If a line is defined which travels backwards, the ends are simply swapped.

Line 1070: 'E' is the error term which will be used to determine whether the line being drawn has deviated enough from the correct course along the Y axis to justify moving one pixel along the Y axis. It is originally set to the difference between the two X coordinates.

Line 1080: The line drawing loop will plot as many pixels as there are positions along the X axis.

Lines 1090–1110: These lines use the HPLOT routine to plot the specified pixel on the screen. The coordinates may need to be swapped for plotting if the line drawing routine has already swapped the X and Y axes for the purposes of calculation.

Lines 1120–1130: The testing of the error term to see whether a move along the Y axis is needed yet.

Lines 2000–2050: This part of the program conducts the overall test by defining a series of lines which form a star on the screen and by calling the line drawing routine to execute each one.

Testing

Simply RUN the program. You should see the high resolution screen set up and series lines drawn which form a star with its centre roughly in the middle of the screen.

Line Plotting — assembly language listing

Having tested the algorithm for line drawing in BASIC, we can now look at the machine code. The method corresponds exactly to what you have just seen and, as far as possible, the variables used have the same names.

```
ADD.  DATA     SOURCE CODE
00             10 PRT
00             20 ORG $C966
C966           30 SYM
C966           70 TEMP    = $14
C966           80 XHI     = $4B
C966           90 XLO     = $4D
C966          100 YHI     = $4F
C966          110 YLO     = $51
C966          120 MODE    = $02
C966          130 SWAPED  = $53
C966          140 DX      = $5F
C966          150 DY      = $61
C966          160 X1      = $63
C966          170 X2      = $65
C966          180 Y1      = $67
C966          190 Y2      = $69
```

```
C966                    200 AVAR    = $6B
C966                    210 BVAR    = $6F
C966                    220 E       = $6D
C966                    230 ABS     = $C8B0
C966                    240 NEGATE  = $C8B4
C966                    250 ADD     = $C8C2
C966                    260 MINUS   = $C8D2
C966                    270 MOVE    = $C8E2
C966                    280 SWAP    = $C8ED
C966                    290 TEST    = $C901
C966                    300 TIMES2  = $C90E
C966                    310 WINDOW  = $C93C
C966                    320 GETWRD  = $C000
C966                    330 GETBYT  = $C006
C966                    340 COMMA   = $AEFD
C966                    370 GETMOD
C966  2006C0            380 JSR GETBYT
C969  C903              390 CMP #3
C96B  B005              400 BCS IQERR
C96D  8502              410 STA MODE
C96F  4CFDAE            420 JMP COMMA
C972                    440 IQERR
C972  20B9CA            450 JSR L007
C975  4C48B2            460 JMP $B248
C978                    490 SETWIN
C978  A900              500 LDA #0
C97A  854D              510 STA XLO
C97C  854E              520 STA XLO+1
C97E  8551              530 STA YLO
C980  8552              540 STA YLO+1
C982  8550              550 STA YHI+1
C984  A9C8              560 LDA #200
C986  854F              570 STA YHI
C988  A940              580 LDA #64
C98A  854B              590 STA XHI
C98C  A901              600 LDA #1
C98E  854C              610 STA XHI+1
C990  60                620 RTS
C991                    660 GETXY2
C991  2000C0            670 JSR GETWRD
C994  2078C9            680 JSR SETWIN
C997  A214              690 LDX #$14
C999  A04B              700 LDY #XHI
C99B  2001C9            710 JSR TEST
C99E  B0D2              720 BCS IQERR
C9A0  A515              730 LDA $15
C9A2  48                740 PHA
C9A3  A514              750 LDA $14
C9A5  48                760 PHA
C9A6  20FDAE            770 JSR COMMA
C9A9  2006C0            780 JSR GETBYT
C9AC  C9C8              790 CMP #200
```

```
C9AE  B0C2      800 BCS IQERR
C9B0  A269      810 LDX #Y2
C9B2  A014      820 LDY #$14
C9B4  20E2C8    830 JSR MOVE
C9B7  68        840 PLA
C9B8  8565      850 STA X2
C9BA  68        860 PLA
C9BB  8566      870 STA X2+1
C9BD  60        880 RTS
C9BE            910 GETXY1
C9BE  2091C9    920 JSR GETXY2
C9C1  A263      930 LDX #X1
C9C3  A065      940 LDY #X2
C9C5  20E2C8    950 JSR MOVE
C9C8  A267      960 LDX #Y1
C9CA  A069      970 LDY #Y2
C9CC  4CE2C8    980 JMP MOVE
C9CF           1010 LINE
C9CF  2066C9   1020 JSR GETMOD
C9D2  20BEC9   1030 JSR GETXY1
C9D5  20FDAE   1040 JSR COMMA
C9D8  A563     1050 LDA X1
C9DA  48       1060 PHA
C9DB  A564     1070 LDA X1+1
C9DD  48       1080 PHA
C9DE  A567     1090 LDA Y1
C9E0  48       1100 PHA
C9E1  A568     1110 LDA Y1+1
C9E3  48       1120 PHA
C9E4  2091C9   1130 JSR GETXY2
C9E7  68       1140 PLA
C9E8  8568     1150 STA Y1+1
C9EA  68       1160 PLA
C9EB  8567     1170 STA Y1
C9ED  68       1180 PLA
C9EE  8564     1190 STA X1+1
C9F0  68       1200 PLA
C9F1  8563     1210 STA X1
C9F3  2078C9   1220 JSR SETWIN
C9F6           1240 DOLINE
C9F6  A900     1250 LDA #0
C9F8  8553     1260 STA SWAPED
C9FA           1270 L000
C9FA  A25F     1290 LDX #DX
C9FC  A065     1300 LDY #X2
C9FE  20E2C8   1310 JSR MOVE
CA01  A063     1320 LDY #X1
CA03  20D2C8   1330 JSR MINUS
CA06  20B0C8   1340 JSR ABS
CA09  A261     1350 LDX #DY
CA0B  A069     1360 LDY #Y2
```

```
CA0D  20E2C8    1370 JSR MOVE
CA10  A067      1380 LDY #Y1
CA12  20D2C8    1390 JSR MINUS
CA15  20B0C8    1400 JSR ABS
CA18  A05F      1420 LDY #DX
CA1A  2001C9    1430 JSR TEST
CA1D  9017      1440 BCC L001
CA1F  F015      1450 BEQ L001
CA21  A9FF      1470 LDA #$FF
CA23  8553      1480 STA SWAPED
CA25  A265      1490 LDX #X2
CA27  A069      1500 LDY #Y2
CA29  20EDC8    1510 JSR SWAP
CA2C  A263      1520 LDX #X1
CA2E  A067      1530 LDY #Y1
CA30  20EDC8    1540 JSR SWAP
CA33  4CFAC9    1550 JMP L000
CA36            1570 L001
CA36  A265      1580 LDX #X2
CA38  A063      1590 LDY #X1
CA3A  2001C9    1600 JSR TEST
CA3D  B00A      1610 BCS L002
CA3F  20EDC8    1630 JSR SWAP
CA42  A267      1640 LDX #Y1
CA44  A069      1650 LDY #Y2
CA46  20EDC8    1660 JSR SWAP
CA49            1680 L002
CA49  A26D      1690 LDX #E
CA4B  A05F      1700 LDY #DX
CA4D  20E2C8    1710 JSR MOVE
CA50  20B4C8    1720 JSR NEGATE
CA53  A269      1740 LDX #Y2
CA55  A067      1750 LDY #Y1
CA57  20D2C8    1760 JSR MINUS
CA5A  A261      1780 LDX #DY
CA5C  200EC9    1790 JSR TIMES2
CA5F  A25F      1800 LDX #DX
CA61  200EC9    1810 JSR TIMES2
CA64            1830 LOOP
CA64  A26B      1850 LDX #AVAR
CA66  A063      1860 LDY #X1
CA68  20E2C8    1870 JSR MOVE
CA6B  A26F      1880 LDX #BVAR
CA6D  A067      1890 LDY #Y1
CA6F  20E2C8    1900 JSR MOVE
CA72  2453      1920 BIT SWAPED
CA74  1005      1930 BPL L003
CA76  A06B      1950 LDY #AVAR
CA78  20EDC8    1960 JSR SWAP
CA7B            1980 L003
CA7B  203CC9    1990 JSR WINDOW
```

```
CA7E  A26D      2010 LDX #E
CA80  A061      2020 LDY #DY
CA82  20C2C8    2030 JSR ADD
CA85  A56E      2050 LDA E+1
CA87  3021      2060 BMI L005
CA89  056D      2070 ORA E
CA8B  F01D      2080 BEQ L005
CA8D  A9FF      2100 LDA #$FF
CA8F  8514      2110 STA TEMP
CA91  8515      2120 STA TEMP+1
CA93  246A      2140 BIT Y2+1
CA95  1005      2150 BPL L004
CA97  A214      2160 LDX #TEMP
CA99  20B4C8    2170 JSR NEGATE
CA9C            2190 L004
CA9C  A267      2200 LDX #Y1
CA9E  A014      2210 LDY #TEMP
CAA0  20D2C8    2220 JSR MINUS
CAA3  A26D      2240 LDX #E
CAA5  A05F      2250 LDY #DX
CAA7  20D2C8    2260 JSR MINUS
CAAA            2280 L005
CAAA  E663      2290 INC X1
CAAC  D002      2300 BNE L006
CAAE  E664      2310 INC X1+1
CAB0            2320 L006
CAB0  A265      2330 LDX #X2
CAB2  A063      2340 LDY #X1
CAB4  2001C9    2350 JSR TEST
CAB7  B0AB      2360 BCS LOOP
CAB9            2380 L007
CAB9  A225      2390 LDX #$25
CABB  A900      2400 LDA #$00
CABD            2410 L008
CABD  954B      2420 STA $4B.X
CABF  CA        2430 DEX
CAC0  10FB      2440 BPL L008
CAC2  A94C      2450 LDA #$4C
CAC4  8554      2460 STA $54
CAC6  60        2470 RTS
CAC7            2480 END
 TOTAL ERRORS IN FILE --- 0


TEMP             14
XHI              4B
XLO              4D
YHI              4F
YLO              51
MODE             02
SWAPED           53
```

```
DX                  5F
DY                  61
X1                  63
X2                  65
Y1                  67
Y2                  69
AVAR                6B
BVAR                6F
E                   6D
ABS                 C8B0
NEGATE              C8B4
ADD                 C8C2
MINUS               C8D2
MOVE                C8E2
SWAP                C8ED
TEST                C901
TIMES2              C90E
WINDOW              C93C
GETWRD              C000
GETBYT              C006
COMMA               AEFD
GETMOD              C966
IQERR               C972
SETWIN              C978
GETXY2              C991
GETXY1              C9BE
LINE                C9CF
DOLINE              C9F6
L000                C9FA
L001                CA36
L002                CA49
LOOP                CA64
L003                CA7B
L004                CA9C
L005                CAAA
L006                CAB0
L007                CAB9
L008                CABD
 TOTAL NUMBER OF SYMBOLS --- 45
```

Machine code

```
ADD   DATA                              CHECKSUM

C966  20 06 C0 C9 03 B0 05 85    3977
C96E  02 4C FD AE 20 B9 CA 4C    4444
C976  48 B2 A9 00 85 4D 85 4E    6C54
C97E  85 51 85 52 85 50 A9 C8    7402
C986  85 4F A9 40 85 4B A9 01    7607
C98E  85 4C 60 20 00 C0 20 78    6738
```

```
C996  C9 A2 14 A0 4B 20 01 C9   9D23
C99E  B0 D2 A5 15 48 A5 14 48   A7B4
C9A6  20 FD AE 20 06 C0 C9 C8   6C8A
C9AE  B0 C2 A2 69 A0 14 20 E2   A9C2
C9B6  C8 68 85 65 68 85 66 60   9B70
C9BE  20 91 C9 A2 63 A0 65 20   5E02
C9C6  E2 C8 A2 67 A0 69 4C E2   C5CE
C9CE  C8 20 66 C9 20 BE C9 20   8AFA
C9D6  FD AE A5 63 48 A5 64 48   CAB4
C9DE  A5 67 48 A5 68 48 20 91   84C1
C9E6  C9 68 85 68 68 85 67 68   9C2A
C9EE  85 64 68 85 63 20 78 C9   7621
C9F6  A9 00 85 53 A2 5F A0 65   7281
C9FE  20 E2 C8 A0 63 20 D2 C8   7184
CA06  20 B0 C8 A2 61 A0 69 20   659A
CA0E  E2 C8 A0 67 20 D2 C8 20   C368
CA16  B0 C8 A0 5F 20 01 C9 90   A716
CA1E  17 F0 15 A9 FF 85 53 A2   6004
CA26  65 A0 69 20 ED C8 A2 63   75CF
CA2E  A0 67 20 ED C8 4C FA C9   86BD
CA36  A2 65 A0 63 20 01 C9 B0   87B6
CA3E  0A 20 ED C8 A2 67 A0 69   3F75
CA46  20 ED C8 A2 6D A0 5F 20   7526
CA4E  E2 C8 20 B4 C8 A2 69 A0   BC7A
CA56  67 20 D2 C8 A2 61 20 0E   6922
CA5E  C9 A2 5F 20 0E C9 A2 6B   A023
CA66  A0 63 20 E2 C8 A2 6F A0   8526
CA6E  67 20 E2 C8 24 53 10 05   66D1
CA76  A0 6B 20 ED C8 20 3C C9   8591
CA7E  A2 6D A0 61 20 C2 C8 A5   8C8D
CA86  6E 30 21 05 6D F0 1D A9   4F7B
CA8E  FF 85 14 85 15 24 6A 10   ADAC
CA96  05 A2 14 20 B4 C8 A2 67   39EB
CA9E  A0 14 20 D2 C8 A2 6D A0   7062
CAA6  5F 20 D2 C8 E6 63 D0 02   689E
CAAE  E6 64 A2 65 A0 63 20 01   AD5D
CAB6  C9 B0 AB A2 25 A9 00 95   B461
CABE  4B CA 10 FB A9 4C 85 54   7186
CAC6  60                        0060
```

Commentary

Lines 70–220: These are the locations for the variables that will be used by the routine, representing spare addresses in the zero page of memory. To the variables used by the BASIC routine are added TEMP, which will be a temporary variable used at certain points, and MODE, which will allow the line to be drawn or erased, as with an individual pixel earlier. As described in the previous routine, we also need parameters for the size of the screen, the variables to define this being HXI, XLO, YHI and YLO.

Lines 370–420: A subroutine to obtain the MODE parameter and test that it is in the range 0–2.

Lines 490–620: Subroutine to set the size of the screen window to 320 across by 200 down. Any lines which fall within this window will be passed by the previous window test routine.

Lines 660–880: A subroutine to pick up the coordinates of the end point of the line. The Y coordinate is tested against the limit of 0–199, since this never changes. The X coordinate is passed to the window test routine, since the width of the window may vary according to whether we are in multi-colour mode or not. If either coordinate is off the screen, an ILLEGAL QUANTITY error is generated.

Lines 910–980: A subroutine to perform the same check for the X1 and Y1 coordinates.

Lines 1000–1220: The main line drawing routine starts here. The previous subroutines are called upon to pick up the coordinates and the MODE. The resulting parameters are returned as numbers stored on the stack. They are pulled off the stack by these lines and placed into the locations specified in the table at the beginning of the routine.

Lines 1250–1260: The variable SWAPED, which indicates whether what would normally be horizontal and vertical have been exchanged for the purposes of calculation, is set to zero.

Lines 1290–1340: These lines perform the equivalent of the BASIC instruction (used in the previous BASIC line drawing routine) DX = ABS (X2 – X1)

Lines 1350–1400: Equivalent to the BASIC instruction DY = ABS (Y2 – Y1)

Lines 1420–1450: Equivalent to the BASIC line IF DX< = DY THEN...

Lines 1470–1480: Set SWAPED to – 1, or $FF, since in 16-bit two's complement format, – 1 = $FFFF.

Lines 1490–1540: Swap X1/Y1 and X2/Y2 using the previously entered subroutines.

Lines 1580–1610: Equivalent to IF X1> X2 THEN GOTO L002.

Line 1630: Swap X1 and X2 only.

Lines 1640–1660: Swap Y1 and Y2 only.

Lines 1690–1720: Equivalent to E = – DX.

Lines 1740–1760: Equivalent to Y2 = Y2 – Y1. Note that this is *not* in the original BASIC program. There is nothing subtle about this command — it is simply that there are not that many storage locations available in the early pages of the memory when we are working with machine code and BASIC. Since the variable Y2 is not going to be used for anything again, the place where it was stored is now a convenient location to store the sign of Y2 – Y1, or YS as it is called in the BASIC program. All that happens is that the two-byte address previously holding the value Y2 will be loaded with the result of Y2 – Y1. The fifteenth, or sign bit, will be a convenient indicator of the value of YS, being set if YS is – 1 and reset if YS is 1.

Lines 1780–1810: From now on, it will not be DX and DY which are required but 2*DX and 2*DY. These lines perform the multiplication in advance.

Lines 1850–1900: Equivalent to A = X1 : B = Y1

Lines 1920–1930: Equivalent to IF SWAPED> 128 THEN GOTO L003. Remember that SWAPED will be either 255 or zero according to whether the X and Y coordinate have been swapped or not.

Lines 1950–1960: These lines swap the values in A and B if necessary.

Line 1990: Calls WINDOW to test and plot the point in question.

Lines 2010–2030: Equivalent to E = E + 2*DY.

Lines 2050–2080: Equivalent to IF E< = 0 THEN GOTO L005.

Lines 2100–2120: In BASIC it is a simple matter to add to Y the value of the sign of Y2 – Y1. All we need to do is to use YS = SGN(Y2 – Y1) and later add YS. There is no such neat equivalent in machine code. To perform the same operation it is simpler to add together two 16-bit numbers, even if one of them (called TEMP in this case) is only going to be plus or minus one. These two lines place the value – 1 into TEMP.

Lines 2140–2170: The contents of the sign bit of Y2 (we are using this to store the result of Y1 – Y2 now) are transferred to the negative flag. The negative flag is now tested and, if positive, a jump is made around the two lines which send TEMP to the negate routine entered earlier, where it will

have its sign changed.

Lines 2200–2220: Equivalent to Y1 = Y1 + YS.

Lines 2240–2260: Equivalent to E = E – 2*DX.

Lines 2290–2360: Equivalent to NEXT X, but notice that to reduce the number of variables we do not bother with a new variable X, we simply use X1 as the loop variable. Equally we could have had a loop FOR X1 = X1 TO X2 in the BASIC program.

Lines 2390–2460: These lines play no part in the actual line drawing. They are needed because this routine has used up so many of the spare locations available for storage when BASIC is running, that we are actually using some locations which will be needed by the BASIC interpreter when our machine code routine returns control to the 64's ROM. These have to be tidied up.

Testing

Take the BASIC line drawing program that you entered earlier and alter it so that the whole section from line 1010 to line 1140 is replaced with:

1010 SYS (51663) 2,X1,Y1,X2,Y2

and lines 2010 and 2020 to:

2010 FOR I1 = 10 TO 310 STEP 20
2020 FOR I2 = 10 TO 190 STEP 20

The result should be a far more complex star pattern which would have taken the earlier program an eternity to print. Do not worry if some of the lines appear to have gaps in them — they are being drawn in inverse mode and they erase each other where they brush.

Syntax

The syntax for LINE is:

SYS (51663) { MODE} , { START POSITION X} , { START POSITION Y} ,{ END POSITION X} ,{ END POSITION Y}

MODE is defined as for the PLOT routine. X must be in the range 0–319 and Y in the range 0–199.

8. HIGH RESOLUTION CIRCLES (CIRCLE)

Having seen the complexity of the line drawing commands, you may not be surprised to discover that a command to draw a circle is also a relatively

complex one. The similarities between the two routines are great, but in the case of drawing a circle the arithmetic required is more complex and, in addition, needs to be considerably more accurate. For that reason, the following routine includes extra routines not merely for 16-bit arithmetic, which we have already entered, but for 15 bit-signed arithmetic *including* provision for 16-bit binary fractions.

The method used to draw a circle works on the basis that the differential of a circle is $-X/Y$. What this means is that if we set out to plot the points around the circle's circumference we will find that the X and Y coordinates of the next point to be plotted can be approximately determined in the following way:

X(next) = X(present) + Y(present)*E
Y(next) = Y(present) – X(next)*E

The mysterious 'E' here is simply a very small number, and the method assumes that the centre of the circle is at position zero on both the X and Y axes.

If it is not obvious to you why such a simple method should produce a circle, consider the following example. We are attempting to draw a circle with a radius of 50 pixels, ie one that will be 100 pixels across. We begin plotting at a point 50 pixels below the centre point or origin, one of four points that we know with certainty will be present in the finished circle, and the plotting of the points on the circle will proceed anticlockwise. The coordinates of this start point are therefore zero on the X axis, since the point is neither to the right nor the left of the origin, and 50 on the Y axis, since we number from the top of the screen downwards.

To plot the next point we use the two expressions above:

X coordinate = 0 + 50*E
Y coordinate = 50 – (50*E)*E

The size of E is not important here, except insofar as the smaller it is, the more accurate the circle (and the longer it will take to draw). What has happened is that the next pixel position has been defined as being to the right of the first by a small amount and down by an even smaller one. This is exactly what we would expect at the bottom of a circle, where the slope is almost flat. Note, however, that even though the change is small the X coordinate is no longer zero. When the next point is plotted, we shall find that Y is decreased by a little more and X increased by a little less, so that the slope upwards increases a little. As we continue plotting points, we shall find that the changes in Y each time will increase and the changes in X decrease until there comes a time when one point is plotted directly above the last, ie the change in X is effectively zero.

If we check the position reached against the origin of the circle, we will find that we have approximately reached a point where:

X = 50
Y = 0

and have drawn one quarter of the circle. When we come to plot the next point we shall find that:

X coordinate = 50 + 0*E
Y coordinate = 0 + (50*E)

In other words, Y is moving a great deal more than X, so the line proceeds straight up, which is what we would expect at this point. As we continue to plot points, we will find that X will begin to decrease as Y moves in a negative direction (above the origin), so the line will begin to curve to the right towards the top of the circle.

We could follow the process all the way round, but you should now have an idea why this simple method will produce an approximation of a circle. Note that it is only an approximation. When the line was at the bottom of the circle it moved to the right and (50*E)*E upwards. When the line was at the far righthand edge of the circle, it moved up and had no inward curve at all at that point. Though the differences are minute, they do add up, and what will be drawn by this method is actually an ellipse, or a circle slightly squashed at the sides. This will be quite noticeable if the radius of the circle is over 50 pixels on the screen and so one extra provision is made before the points of the circle are plotted. This is to multiply one coordinate by 1 + E and the other by 1 − E (depending on which way the circle needs to be 'squashed'), values which differ so little from 1 that all they do is to correct the slight tendency for the circle to become an ellipse.

Circle — BASIC listing

```
1 GOTO 3
2 SAVE "@0:CIRCLE DRAW",8 : VERIFY "CIRC
LE DRAW",8 : STOP
3 REM
10 POKE 56,12 : CLR
20 BITMAP = 8192
30 COLOUR = 3072
40 SYS (51044) BITMAP,COLOUR
70 SYS 51003
80 SYS 51116
90 GOSUB 2000
170 SYS 50944
180 END
1000 REM DRAW CIRCLE
1010 X1 = 0 : Y1 = R
1020 E = 0.5
1030 E = E*2 : IF E<R THEN 1030
1040 E = E*4
1100 FOR T = 1 TO E*7
```

```
1110 A = X+X1*(1-1/E)
1111 B = Y+Y1*(1+1/E)
1115 IF A>319 OR A<0 OR B>199 OR B<0 THE
N 1130
1120 SYS (51304) 0,A,B
1130 X1 = X1+Y1/E
1140 Y1 = Y1-X1/E
1150 NEXT T
1160 RETURN
2000 REM TEST CIRCLE DRAW
2010 X = 160 : Y = 100
2020 FOR R = 10 TO 130  STEP 40
2030 GOSUB 1000
2040 NEXT R
2060 GET T$ : IF T$="" THEN 2060
2070 RETURN
```

Commentary

Lines 10–180: As in line drawing program.

Lines 1000–1160: The routine to plot the points of a circle. To work, it must be supplied with the X and Y coordinates of the origin of the circle and the radius. The first point does not need calculation since it can be defined from the centre coordinates and the radius, but the process of multiplying by 1 + E and 1 – E is carried out before plotting. In this case, E is set to at least four times the radius of the circle in pixels and then 1/E used for calculations. This multiplication by four ensures that there will be enough points plotted so that the circle will always appear as a solid line.

Lines 2000–2070: The control routine. This defines a series of five circles of increasing radius based on a common origin.

Testing

Lines 2000–2070 are intended to make the program self-testing. Once again, the BASIC program is designed, based on the assumption that it will be RUN with the previous high resolution commands resident in the memory. If the routine is correct it will draw four concentric circles.

Circle — assembly language listing

```
ADD.   DATA      SOURCE CODE
00               10 PRT
00               20 ORG $CAC7
CAC7             30 SYM
CAC7             80 TEMP   = $5E
CAC7             90 X1     = $56
CAC7            100 X2     = $63
CAC7            110 Y1     = $5A
CAC7            120 Y2     = $67
```

```
CAC7                130 AVAR   = $6B
CAC7                140 BVAR   = $6F
CAC7                150 E      = $6D
CAC7                160 T      = $14
CAC7                170 COUNT  = $53
CAC7                180 NEGATE = $C8B4
CAC7                190 ADD    = $C8C2
CAC7                200 MINUS  = $C8D2
CAC7                210 MOVE   = $C8E2
CAC7                220 TEST   = $C901
CAC7                230 TIMES2 = $C90E
CAC7                240 WINDOW = $C93C
CAC7                250 COMMA  = $AEFD
CAC7                260 GETMOD = $C966
CAC7                270 EXIT   = $CAB9
CAC7                280 GETWRD = $C000
CAC7                290 GETXY2 = $C991
CAC7                340 DDIV
CAC7  A453          350 LDY COUNT
CAC9                360 L000
CAC9  B503          380 LDA 3.X
CACB  0A            390 ASL A
CACC  7603          400 ROR 3.X
CACE  7602          410 ROR 2.X
CAD0  7601          420 ROR 1.X
CAD2  7600          430 ROR 0.X
CAD4  88            440 DEY
CAD5  D0F2          450 BNE L000
CAD7  60            460 RTS
CAD8                490 DMOVE
CAD8  20E2C8        500 JSR MOVE
CADB  E8            510 INX
CADC  E8            520 INX
CADD  C8            530 INY
CADE  C8            540 INY
CADF  20E2C8        550 JSR MOVE
CAE2                560 LEAVE
CAE2  CA            570 DEX
CAE3  CA            580 DEX
CAE4  88            590 DEY
CAE5  88            600 DEY
CAE6  60            610 RTS
CAE7                640 DADD
CAE7  20C2C8        650 JSR ADD
CAEA  E8            660 INX
CAEB  E8            670 INX
CAEC  C8            680 INY
CAED  C8            690 INY
CAEE  20C3C8        700 JSR ADD+1
CAF1  4CE2CA        710 JMP LEAVE
CAF4                740 DMINUS
```

```
CAF4  20D2C8     750 JSR MINUS
CAF7  E8         760 INX
CAF8  E8         770 INX
CAF9  C8         780 INY
CAFA  C8         790 INY
CAFB  20D3C8     800 JSR MINUS+1
CAFE  4CE2CA     810 JMP LEAVE
CB01             840 DNEG
CB01  20B4C8     850 JSR NEGATE
CB04  E8         860 INX
CB05  E8         870 INX
CB06  C8         880 INY
CB07  C8         890 INY
CB08  20B5C8     900 JSR NEGATE+1
CB0B  4CE2CA     910 JMP LEAVE
CB0E             940 IQERR
CB0E  20B9CA     950 JSR EXIT
CB11  4C48B2     960 JMP $B248
CB14             990 CIRCLE
CB14  2066C9    1010 JSR GETMOD
CB17  2000C0    1030 JSR GETWRD
CB1A  A515      1040 LDA T+1
CB1C  C902      1050 CMP #2
CB1E  B0EE      1060 BCS IQERR
CB20  48        1070 PHA
CB21  A514      1080 LDA T
CB23  48        1090 PHA
CB24  20FDAE    1100 JSR COMMA
CB27  2091C9    1120 JSR GETXY2
CB2A  68        1130 PLA
CB2B  855C      1140 STA Y1+2
CB2D  68        1150 PLA
CB2E  855D      1160 STA Y1+3
CB30            1180 DOCIR
CB30  A900      1200 LDA #0
CB32  8556      1210 STA X1
CB34  8557      1220 STA X1+1
CB36  8558      1230 STA X1+2
CB38  8559      1240 STA X1+3
CB3A  855A      1250 STA Y1
CB3C  855B      1260 STA Y1+1
CB3E  8553      1270 STA COUNT
CB40  8563      1280 STA X2
CB42  8564      1290 STA X2+1
CB44  8567      1300 STA Y2
CB46  8568      1310 STA Y2+1
CB48  856E      1320 STA E+1
CB4A  A901      1330 LDA #1
CB4C  856D      1340 STA E
CB4E  A26D      1360 LDX #E
CB50  A05C      1370 LDY #Y1+2
```

```
CB52                1380 L001
CB52  2001C9        1390 JSR TEST
CB55  B007          1400 BCS L002
CB57  200EC9        1420 JSR TIMES2
CB5A  E653          1430 INC COUNT
CB5C  D0F4          1440 BNE L001
CB5E                1450 L002
CB5E  200EC9        1500 JSR TIMES2
CB61  200EC9        1510 JSR TIMES2
CB64  200EC9        1520 JSR TIMES2
CB67                1540 LOOP
CB67  A25E          1560 LDX #TEMP
CB69  A056          1570 LDY #X1
CB6B  20D8CA        1580 JSR DMOVE
CB6E  20C7CA        1590 JSR DDIV
CB71  2001CB        1600 JSR DNEG
CB74  A056          1610 LDY #X1
CB76  20E7CA        1620 JSR DADD
CB79  A063          1630 LDY #X2
CB7B  20E7CA        1640 JSR DADD
CB7E  A26B          1650 LDX #AVAR
CB80  A060          1660 LDY #TEMP+2
CB82  20E2C8        1670 JSR MOVE
CB85  A25E          1690 LDX #TEMP
CB87  A05A          1700 LDY #Y1
CB89  20D8CA        1710 JSR DMOVE
CB8C  20C7CA        1720 JSR DDIV
CB8F  A05A          1730 LDY #Y1
CB91  20E7CA        1740 JSR DADD
CB94  A067          1750 LDY #Y2
CB96  20E7CA        1760 JSR DADD
CB99  A26F          1770 LDX #BVAR
CB9B  A060          1780 LDY #TEMP+2
CB9D  20E2C8        1790 JSR MOVE
CBA0  203CC9        1810 JSR WINDOW
CBA3  A25E          1830 LDX #TEMP
CBA5  A05A          1840 LDY #Y1
CBA7  20D8CA        1850 JSR DMOVE
CBAA  20C7CA        1860 JSR DDIV
CBAD  A256          1870 LDX #X1
CBAF  A05E          1880 LDY #TEMP
CBB1  20E7CA        1890 JSR DADD
CBB4  A25E          1910 LDX #TEMP
CBB6  A056          1920 LDY #X1
CBB8  20D8CA        1930 JSR DMOVE
CBBB  20C7CA        1940 JSR DDIV
CBBE  A25A          1950 LDX #Y1
CBC0  A05E          1960 LDY #TEMP
CBC2  20F4CA        1970 JSR DMINUS
CBC5  A56D          1990 LDA E
CBC7  D002          2000 BNE L003
CBC9  C66E          2010 DEC E+1
```

```
CBCB              2020 L003
CBCB  C66D        2030 DEC E
CBCD  A56D        2040 LDA E
CBCF  056E        2050 ORA E+1
CBD1  D094        2060 BNE LOOP
CBD3  4CB9CA      2080 JMP EXIT
CBD6              2090 END
 TOTAL ERRORS IN FILE --- 0


TEMP              5E
X1                56
X2                63
Y1                5A
Y2                67
AVAR              6B
BVAR              6F
E                 6D
T                 14
COUNT             53
NEGATE            C8B4
ADD               C8C2
MINUS             C8D2
MOVE              C8E2
TEST              C901
TIMES2            C90E
WINDOW            C93C
COMMA             AEFD
GETMOD            C966
EXIT              CAB9
GETWRD            C000
GETXY2            C991
DDIV              CAC7
L000              CAC9
DMOVE             CAD8
LEAVE             CAE2
DADD              CAE7
DMINUS            CAF4
DNEG              CB01
IQERR             CB0E
CIRCLE            CB14
DOCIR             CB30
L001              CB52
L002              CB5E
LOOP              CB67
L003              CBCB
 TOTAL NUMBER OF SYMBOLS --- 36
```

Machine code

```
ADD   DATA                          CHECKSUM
CAC7  A4 53 B5 03 0A 76 03 76        8034
```

```
CACF  02 76 01 76 00 88 D0 F2   2AB2
CAD7  60 20 E2 C8 E8 E8 C8 C8   6DF8
CADF  20 E2 C8 CA CA 88 88 60   7800
CAE7  20 C2 C8 E8 E8 C8 C8 20   7410
CAEF  C3 C8 4C E2 CA 20 D2 C8   B45C
CAF7  E8 E8 C8 C8 20 D3 C8 4C   D9A8
CAFF  E2 CA 20 B4 C8 E8 E8 C8   BF38
CB07  C8 20 B5 C8 4C E2 CA 20   96BC
CB0F  B9 CA 4C 48 B2 20 66 C9   A4A5
CB17  20 00 C0 A5 15 C9 02 B0   36D0
CB1F  EE 48 A5 14 48 20 FD AE   A448
CB27  20 91 C9 68 85 5C 68 85   5ACD
CB2F  5D A9 00 85 56 85 57 85   6707
CB37  58 85 59 85 5A 85 5B 85   66CF
CB3F  53 85 63 85 64 85 67 85   65F7
CB47  68 85 6E A9 01 85 6D A2   7128
CB4F  6D A0 5C 20 01 C9 B0 07   7093
CB57  20 0E C9 E6 53 D0 F4 20   42E0
CB5F  0E C9 20 0E C9 20 0E C9   45CD
CB67  A2 5E A0 56 20 D8 CA 20   87F4
CB6F  C7 CA 20 01 CB A0 56 20   A3B4
CB77  E7 CA A0 63 20 E7 CA A2   C702
CB7F  6B A0 60 20 E2 C8 A2 5E   7752
CB87  A0 5A 20 D8 CA 20 C7 CA   8128
CB8F  A0 5A 20 E7 CA A0 67 20   82AE
CB97  E7 CA A2 6F A0 60 20 E2   C8D2
CB9F  C8 20 3C C9 A2 5E A0 5A   8832
CBA7  20 D8 CA 20 C7 CA A2 56   6C3A
CBAF  A0 5E 20 E7 CA A2 5E A0   8424
CBB7  56 20 D8 CA 20 C7 CA A2   60F2
CBBF  5A A0 5E 20 F4 CA A5 6D   6F3F
CBC7  D0 02 C6 6E C6 6D A5 6D   91BB
CBCF  05 6E D0 94 4C B9 CA      240C
```

Commentary

Lines 80–170: The list of locations for variables to be held in zero-page memory. The variable COUNT will be used to hold the power of two on which E will be based. Defining E as a power of two will allow us to perform the division E without writing another lengthy arithmetic routine since all that will be required is a series of shifts to the right.

Lines 340–460: Subroutine to divide the four-byte number pointed to the X register by 2^value held in the variable location COUNT. The division is achieved by loading the value of COUNT into the Y register and using this as a loop counter for a series of shifts to the right.

Lines 490–610: Equivalent to the 16-bit routine MOVE (which it uses) but deals with a four byte location.

Lines 640–710: 32-bit equivalent of the previously entered add routine.

Lines 740–810: Equivalent to the previously entered minus routine.

Lines 840–910: Equivalent to the previously entered negate routine.

Lines 950–960: These lines are called whenever an illegal quantity error is detected. They first of all execute the housekeeping routine from the line drawing command, tidying up zero page memory before the return to BASIC.

Line 1010: Obtains the MODE for plotting.

Line 1030: Obtains the value for the radius. This value is tested against a maximum of 511 by sampling the most significant bit. CIRCLE is capable of handling circles which are too large to fit on the screen in their entirety, leaving it to the previously entered window routine to avoid the plotting of points which do not fall on the screen. The limit of 511 is imposed by the fact that the radius is going to be multiplied by 32 for greater accuracy of calculation — any value greater than 511 would no longer fit into a 15-bit signed number.

Lines 1120–1160: Obtain the coordinates of the origin and use the Y coordinate of the origin to set up the Y coordinate of the first point to be plotted (Y1).

Lines 1190–1340: The remaining variables are initialised to zero, with the exception of E, which is set to one.

Lines 1360–1440: Equivalent to the BASIC statement E = E*2 : IF E< R THEN GOTO LOOP.

Lines 1470–1520: Multiply E further by a sufficient value to ensure that the circle will be plotted as a solid line.

Lines 1560–1790: A and B, the coordinates required by the plotting routine, are calculated on the basis of the method described above. Note that X and Y values arrived at do not include any allowance for the coordinates of the origin on the screen, they are simply positioned relative to the origin. The X and Y coordinates of the origin must be added in before anything is plotted.

Line 1810: A call to the window routine which will in turn call up HPLOT

to place the specified pixel on the screen.

Lines 1830–1970: These lines calculate the next X and Y positions according to the method explained above.

Lines 1990–2060: The variable E is used as an index for the main loop since its working value has been transferred to COUNT.

Line 2080: A call to the housekeeping routine to tidy up zero-page memory before leaving the circle routine.

Testing

As with the line drawing routine entered earlier, the simplest way to test CIRCLE is to replace the part of the BASIC program which drew the circles with a call to the machine code routine. Lines 1010–1150 of the BASIC program are therefore reduced to:

1010 SYS (51988) MODE,R,X,Y

and the following two lines added:

2015 FOR MODE = 0 TO 1
2040 NEXT R,MODE

The program should now work as it did previously in BASIC, but considerably faster — then erase the circles.

Syntax and notes on use

The syntax for CIRCLE is:

SYS (51988) { MODE} ,{ RADIUS} ,{ CENTRE X COORDINATE},
{ CENTRE Y COORDINATE}

MODE is as in the line drawing and plotting routines. RADIUS may be any value in the range 0–511. CENTRE X COORDINATE may be in the range 0–319. CENTRE Y COORDINATE may be in the range 0–191.

Note that it is not really practical to use CIRCLE with mode 2, the invert mode. So much overplotting takes place during the drawing of the circle that it is partially erasing itself as it is drawn.

9. HIGH RESOLUTION TEXT (TEXT and PUTCHR)

Now that we have a series of commands which make the use of the high resolution screen a practical proposition in everyday programming, we can put the icing on the cake by removing one of the most annoying limitations on the use of high resolution modes on many microcomputers — the inability to print ordinary text on the high resolution screen.

The high resolution text capability is limited in that it allows characters to be printed only in the 1000 character cells you would find on the normal low resolution screen — you cannot print a character beginning halfway across or down a normal character position.

The machine code version of the TEXT command comes in two sections, one to control the taking of the text from the BASIC line and the other to do the main task of placing the text on the screen. The procedure, once a letter has been picked up from a string to be printed, is as follows:

1) Convert the ASCII code of the character to its screen code.
2) Test to see if the reverse flag is set, if so set bit seven of the screen code to convert the character to its inverse.
3) Calculate the position of the data for the character in the ROM.
4) Switch off the interrupts and 'switch in' the ROM character data.
5) Copy the bytes which make up the character shape into the correct character position on the screen.
6) Switch out the character ROM and switch the interrupts back on.
7) Set the colour of the character square to the current printing colour.
8) Move the low resolution cursor one place to the left. If the cursor moves off the bottom of the screen, move it back to the top lefthand corner.

Text — BASIC listing

```
29000 REM*******************************
29001 REM HIGH RESOLUTION TEXT
29002 REM*******************************
29010 HT$ = "THIS IS HIGH RESOLUTION TEX
T"
29015 HH=0 : HV=0
29020 FOR I=1 TO LEN(HT$)
29030 TT$=MID$(HT$,I,1)
29040 GOSUB 30000
29050 GOSUB 27000
29060 GOSUB 31000
29100 NEXT I
29200 RETURN
30000 REM*******************************
30001 REM ASCII TO SCREEN CODE
30002 REM*******************************
30010 TT=ASC(TT$)
30020 IF TT=255 THEN TT = 126
30025 IF TT=>192 AND TT<=223 THEN TT=TT-
96
30030 IF TT=>224 AND TT<=254 THEN TT=TT-
64
30040 IF TT<32 THEN  TT=-1 : GOTO 30500
30050 IF TT=>64 AND TT<=95 THEN TT=TT-64
 : GOTO 30500
```

```
30060 IF TT=>96 AND TT<=127 THEN TT=TT-3
2 : GOTO 30500
30070 IF TT=>160 AND TT<=191 THEN TT=TT-
64 : GOTO 30500
30500 RETURN
31000 REM*******************************
31001 REM PLACE ON SCREEN
31002 REM*******************************
31010 CP=53248+8*TT
31020 POKE 56334,PEEK(56334) AND 254
31030 POKE 1,PEEK(1) AND 251
31100 FOR J=0 TO 7
31110 POKE C1 + J, PEEK(CP+J)
31120 NEXT J
31130 GOSUB 28500
31200 POKE 1,PEEK(1) OR 4
31210 POKE 56334, PEEK(56334) OR 1
31300 HH=HH+8 : IF HH>319 THEN HH=0 : HV
= HV+8
31310 IF HV>199 THEN HV=0
31500 RETURN
```

Commentary

Lines 29010–29015: For the purposes of testing, the string to be printed is supplied in the routine itself, along with its position in high resolution pixels.

Lines 29020–29100: The main loop, which calls up the various working modules in turn.

Line 30020: The PI character is an eccentric one which appears in two places in the character set. One of the positions is 255 and if this is encountered it is changed to the alternative value 126 which can be dealt with by the lines below.

Line 30025: This routine does not deal with control characters of any kind. If they are encountered in a string to be printed they are ignored. This line flags a control character by setting TT to an impossible value.

Lines 30030–30070: These lines translate the ASCII code to the screen code. There is no particular principle to them, it is simply that they do the job.

Line 31005: The character is a control character if TT is minus one.

Line 31080: This line calculates the start point in the ROM for the data for the character TT.

Lines 31020–31030: These are lines you should remember from the routine to copy the character set. Their effect is to switch off the interrupts and to map the character ROM into memory in its correct location.

Lines 31100–31120: The eight bytes of character data are copied into the eight bytes of the character position specified by the variable C1 (which records the first byte of the character position).

Lines 31200–31210: The character ROM is switched out and the interrupts back on.

Lines 31300–31310: The high resolution text position is incremented by eight pixels, ie one character square. If the resultant character square goes off the end of a line, then the position is wrapped around to the beginning of the next line. If this results in a position off the bottom of the screen, it is wrapped around to the beginning of the *first* line on the screen.

Text — assembly language listing

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $CBDC
CBDC              30 SYM
CBDC              80 GETSTR = $C089
CBDC              90 ADD    = $006B
CBDC             100 LENGTH = $006A
CBDC             110 PUTCHR = $C660
CBDC             140 TEXT
CBDC  2089C0     150 JSR GETSTR
CBDF             160 L000
CBDF  A66A       170 LDX LENGTH
CBE1  F011       180 BEQ L001
CBE3  A000       190 LDY #0
CBE5  B16B       200 LDA (ADD).Y
CBE7  2060C6     210 JSR PUTCHR
CBEA  C66A       220 DEC LENGTH
CBEC  E66B       230 INC ADD
CBEE  D0EF       240 BNE L000
CBF0  E66C       250 INC ADD+1
CBF2  D0EB       260 BNE L000
CBF4             270 L001
CBF4  60         280 RTS
CBF5             290 END

 TOTAL ERRORS IN FILE --- 0


GETSTR           C089
ADD              6B
```

```
LENGTH          6A
PUTCHR          C660
TEXT            CBDC
L000            CBDF
L001            CBF4
 TOTAL NUMBER OF SYMBOLS --- 7
```

Machine code

```
ADD.   DATA                          CHECKSUM

CBDC   20 89 C0 A6 6A F0 11 A0       5C72
CBE4   00 B1 6B 20 60 C6 C6 6A       43AE
CBEC   E6 6B D0 EF E6 6C D0 EB       C21B
CBF4   60                            0060
```

Commentary

Line 150: The GETSTR routine, one of the first you entered, is used to pick up the string to be printed.

Lines 170–180: The length of the string has been stored by GETSTR in the variable LENGTH and this is tested to see that it has not reached zero.

Lines 190–200: The start point of the string in the BASIC program has been stored by GETSTR in the variable ADD. This variable is now used, together with the Y register, to specify a character in the string and to load it into the accumulator.

Line 210: The following routine, PUTCHR, is called up to print the character on the screen.

Lines 220–260: The variable LENGTH is decremented and ADD increased by one. The loop is now executed again, with the next character being picked up for passing to PUTCHR.

PutChr — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C660
C660               30 SYM
C660              110 REVFLG = $00C7
C660              120 PI     = $FF
C660              130 HBIT   = $02DF
C660              140 HLINE  = $02DC
C660              150 GETCUR = $C7E7
C660              160 TEMP   = $0014
C660              170 CURSOR = $00F3
C660              180 LOCOL  = $00D3
```

```
C660            190 LOLINE = $00D6
C660            200 LOCUR  = $00D1
C660            210 COLOUR = $0286
C660            220 CLR    = $C7AC
C660            230 HOME   = $C7DC
C660            240 CHR    = $00D7
C660            270 PUTCHR
C660  85D7      290 STA CHR
C662  C9FF      310 CMP #PI
C664  D002      320 BNE L000
C666  A95E      330 LDA ##$5E
C668            340 L000
C668  C993      360 CMP #147
C66A  F07E      370 BEQ DOCLR
C66C  C913      380 CMP #19
C66E  F07D      390 BEQ DOHOME
C670  297F      400 AND ##$7F
C672  C920      410 CMP ##$20
C674  907A      420 BCC L008
C676  24D7      440 BIT CHR
C678  1004      450 BPL L001
C67A  0940      460 ORA ##$40
C67C  D00A      470 BNE L003
C67E            480 L001
C67E  5008      490 BVC L003
C680  C960      500 CMP ##$60
C682  B002      510 BCS L002
C684  29BF      520 AND ##$BF
C686            530 L002
C686  29DF      540 AND ##$DF
C688            550 L003
C688  A6C7      570 LDX REVFLG
C68A  F002      580 BEQ L004
C68C  0980      590 ORA ##$80
C68E            600 L004
C68E  8514      620 STA TEMP
C690  A91A      630 LDA ##$1A
C692  A003      640 LDY #3
C694            650 L005
C694  0614      660 ASL TEMP
C696  2A        670 ROL A
C697  88        680 DEY
C698  D0FA      690 BNE L005
C69A  8515      700 STA TEMP+1
C69C  20E7C7    710 JSR GETCUR
C69F  AD0EDC    730 LDA 56334
C6A2  29FE      740 AND #254
C6A4  8D0EDC    750 STA 56334
C6A7  A501      760 LDA 1
C6A9  29FB      770 AND #251
C6AB  8501      780 STA 1
```

```
C6AD  A5F3       800 LDA CURSOR
C6AF  29F8       810 AND ##$F8
C6B1  85F3       820 STA CURSOR
C6B3  A007       830 LDY #7
C6B5             840 L006
C6B5  B114       850 LDA (TEMP).Y
C6B7  91F3       860 STA (CURSOR).Y
C6B9  88         870 DEY
C6BA  10F9       880 BPL L006
C6BC  A501       900 LDA 1
C6BE  0904       910 ORA #4
C6C0  8501       920 STA 1
C6C2  EE0EDC     930 INC 56334
C6C5  A4D3       950 LDY LOCOL
C6C7  B1D1       960 LDA (LOCUR).Y
C6C9  290F       970 AND ##$F
C6CB  8514       980 STA TEMP
C6CD  AD8602     990 LDA COLOUR
C6D0  0A        1000 ASL A
C6D1  0A        1010 ASL A
C6D2  0A        1020 ASL A
C6D3  0A        1030 ASL A
C6D4  0514      1040 ORA TEMP
C6D6  91D1      1050 STA (LOCUR).Y
C6D8  A6D6      1070 LDX LOLINE
C6DA  C8        1080 INY
C6DB  C040      1090 CPY ##$40
C6DD  9007      1100 BCC L007
C6DF  A000      1110 LDY #0
C6E1  E8        1120 INX
C6E2  E025      1130 CPX ##$25
C6E4  B007      1140 BCS DOHOME
C6E6            1150 L007
C6E6  18        1160 CLC
C6E7  4CF0FF    1170 JMP $FFF0
C6EA            1180 DOCLR
C6EA  4CACC7    1190 JMP CLR
C6ED            1200 DOHOME
C6ED  4CDCC7    1210 JMP HOME
C6F0            1220 L008
C6F0  60        1230 RTS
C6F1            1240 END
 TOTAL ERRORS IN FILE --- 0


REVFLG          C7
PI              FF
HBIT            2DF
HLINE           2DC
GETCUR          C7E7
TEMP            14
```

```
CURSOR              F3
LOCOL               D3
LOLINE              D6
LOCUR               D1
COLOUR              286
CLR                 C7AC
HOME                C7DC
CHR                 D7
PUTCHR              C660
L000                C668
L001                C67E
L002                C686
L003                C688
L004                C68E
L005                C694
L006                C6B5
L007                C6E6
DOCLR               C6EA
DOHOME              C6ED
L008                C6F0
 TOTAL NUMBER OF SYMBOLS --- 26
```

Machine code

```
ADD   DATA                          CHECKSUM

C660  85 D7 C9 FF D0 02 A9 5E   A988
C668  C9 93 F0 7E C9 13 F0 7D   B811
C670  29 7F C9 20 90 7A 24 D7   56E7
C678  10 04 09 40 D0 0A 50 08   1570
C680  C9 60 B0 02 29 BF 29 DF   9815
C688  A6 C7 F0 02 09 80 85 14   A646
C690  A9 1A A0 03 06 14 2A 88   708C
C698  D0 FA 85 15 20 E7 C7 AD   BF47
C6A0  0E DC 29 FE 8D 0E DC A5   59FD
C6A8  01 29 FB 85 01 A5 F3 29   371B
C6B0  F8 85 F3 A0 07 B1 14 91   C955
C6B8  F3 88 10 F9 A5 01 09 04   B252
C6C0  85 01 EE 0E DC A4 D3 B1   6D27
C6C8  D1 29 0F 85 14 AD 86 02   8152
C6D0  0A 0A 0A 0A 05 14 91 D1   0BCB
C6D8  A6 D6 C8 C0 40 90 07 A0   B26E
C6E0  00 E8 E0 25 B0 07 18 4C   5E68
C6E8  F0 FF 4C AC C7 4C DC C7   D5E7
C6F0  60                        0060
```

Commentary

Lines 290–550: These lines, which will be described in more detail in what follows, have the overall purpose of taking the character code of the next character to be printed and converting it to a screen code. Control characters will be ignored by PUTCHR with the exception of CLEAR and

HOME, which will be executed as HCLEAR and HHOME.

Lines 290–330: Having stored the character code in the variable CHR, these lines deal with the unique case of PI which sits in the anomalous position of 255 in the character set and fits none of the rules for conversion to screen codes.

Lines 360–420: Printing the control codes, such as those concerned with colour, presents difficulties so, as mentioned earlier, we limit the control codes which will be acted upon to CLEAR and HOME. These lines detect the character codes for the two allowed controls and, if they are found, call up the high resolution versions of the commands. The method used is to AND the ASCII code for the character with 127 and then to compare the result with 32. The effect of this is to detect any character which has a code of 0–31 or 128–159. If the character code *does* fall into either of these ranges the routine jumps to the end and the character is not printed.

Lines 430–540: There is no particular logic to the conversion of standard ASCII character codes to the screen codes used by the 64. These lines follow one or two rules of thumb which will make the conversion.

Lines 570–590: The routine is capable of detecting whether string contains the RVS ON control character by sampling the 'reverse flag' at this point. If the flag is set then the screen code to be printed is ORed with 128, thus setting bit seven of the screen code value.

Lines 620–700: We now need to obtain the shape of the character from the character generator ROM. To do this, the screen code is first stored in the variable TEMP. The accumulator is then loaded with the value $1A. Taking the two values in TEMP (least significant byte) and the accumulator (most significant byte), we now have a two-byte value equal to one-eighth of the address in memory of the start of the specified character in the character generator ROM — the screen code of a character times eight is equal to the start position of the data for that character in the screen generator ROM. All that these particular lines do is to take the two-byte number contained in TEMP and the accumulator and shift it left three times, thus multiplying by eight. The value in the accumulator is then tranferred to TEMP + 1, so that the full address of the start of the character data is now held in the two-byte value at TEMP.

Lines 705–710: A call to GETCUR to calculate the correct start position in memory of the current character position on the high resolution screen. To ensure that the character does not start halfway across or down a character position, HIBIT and HILINE are first set to zero.

Lines 720–780: The character generator ROM is not part of normal memory, as we found when copying character data for use in modifying the character set. The procedure for mapping character data into RAM is the same as that used in CCOPY, earlier.

Lines 830–880: Using the locations for the character data (stored in TEMP) and the screen character position (stored in CURSOR), all that needs to be done now is to transfer the eight bytes which define the shape of the character on to the screen.

Lines 900–930: The interrupts are turned back on and the memory returned to its normal state.

Lines 950–1050: These lines obtain the current printing colour from a system register and then place that colour into the upper four bits of the colour memory byte corresponding to the current character position.

Lines 1070–1170: The low resolution cursor is moved one place to the right, so that the next character will be printed in the correct position. Note that if you go off the bottom righthand corner of the screen it will not scroll, the print position will simply wrap around to the top lefthand corner.

Lines 1190–1210: Execute the HCLEAR and HHOME controls if they are detected in the string.

Testing

BASIC: The routine is self-testing. Change line 22700 to read:

22700 GOSUB 29000

Instead of merely clearing the screen, it should now print 'THIS IS HIGH RESOLUTION TEXT' on the top line of the screen.

MACHINE CODE: The following short BASIC program should be all you need to ensure that HCLEAR is working properly:

```
1 GOTO 3
2 SAVE"@0:TEXT TEST",8 : VERIFY "0:TEXT
TEST",8 : STOP
3 REM
20 POKE56,12:CLR
30 COLOUR=3072
40 BITMAP=8192
50 SYS (51044) BITMAP,COLOUR
70 SYS 51003
```

```
75 SYS (52188) "[CLR]"
80 FOR I = 0 TO 1000 : NEXT
90 SYS (52188) "ABCDEFGHIJKLMNOPQRSTUVWX
YZ1234567890+-\@*:;/.,_!#$%&'()^"
100 GET T$ : IF T$="" THEN 100
170 SYS 50944
180 END
```

RUNning the program should clear the screen (line 75) and then print the string contained in line 90.

Syntax

The syntax for TEXT is:

SYS (52188) { STRING EXPRESSION}

where the string expression may be any valid string expression not exceeding 255 characters. All control characters are excluded except for CLR and HOME.

CHAPTER 4
Sprite Commands

1. SPRITES EXPLAINED

The major reason for the power and flexibility of the 64's sprites is simply that they are in many ways not part of the 64's normal system. Sprites appear on the screen but they are never a part of screen memory. If that sounds strange, think for a moment of the enormous problems which would arise in trying to place the same shape into the low resolution, high resolution and multicolour high resolution mode screens, each with their different configuration. Think, too, of the problems that would result as the sprite had to be subjected to complex manipulation to simulate movement or to change its size.

In fact, none of these problems arise because sprites are an illusion, a signal which the VIC chip mixes with the contents of the screen *after* it has made up the picture in whatever mode is in force at the moment. Rather like the light of a torch, played across a wall, seeming to be part of it and yet totally free of it, the sprite is free of the normal constraints of the screen because it is not part of the screen as far as the 64 is concerned.

It follows from all of this that the major questions to be answered when it comes to understanding how to use sprites are:

1) Where does the VIC chip get the information which defines the sprite design?
2) How does it know *when* to place a sprite on the screen?
3) How does it know *where* to place a sprite on the screen?
4) How is the sprite colour defined?
5) How is sprite size defined?

1. Defining a sprite

A basic sprite is a box with dimensions in terms of high resolution pixels of 24 across and 21 down. This amounts to 504 pixels and, since we already know that an individual pixel can be represented by no more than a single bit, 504/8 = 63 bytes. To define a sprite, the VIC chip needs much the same kind of information as in the case of a character, namely a series of bytes which define, in ones and zeros, which particular pixels in the square are to be on or off. What we are trying to arrive at is a series of values which will produce something like this:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0   ←
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

This series of ones and zeros defines, if you look closely, a box with a hole cut in the centre. The question is, how to translate the picture into values which can be stored in memory. The clue is the number of positions across the sprite, ie 24. Clearly, 24 positions can be divided into three groups of eight, so that each line across a sprite can be represented by three bytes. In the case of the line marked with an arrow, the three bytes would be:

00000111 = 7
00000000 = 0
11100000 = 224

If each line of the sprite can be defined by three bytes, the whole sprite should be definable in 21*3 bytes, and this is exactly what happens. When told to place a sprite on the screen, the VIC chip looks for a block of 63 bytes of data and translates it into the picture which appears on the screen.

To do this, the VIC chip must be clearly told *where* to find the data, since any 63 bytes are capable of being translated even though most of them will produce garbage. This task is left to a series of eight pointers called, unsurprisingly, the sprite pointers. Sprite pointers have a fixed position, not in the memory as a whole, nor even in the current video bank, but in relation to the low resolution screen memory (colour memory in high resolution). The low resolution screen, as we have seen, requires only 1000 bytes of memory, even though 1K (1024 bytes) is allocated to it. There are eight

possible sprites and their pointers will be found in the last eight, unused bytes of the 1K block.

In normal circumstances this will place them at 2040–2047, with the last digit of the address conveniently corresponding to the number of the sprite to which the byte acts as a pointer. Since a single byte only is used, it is clear that what is being stored is not a complete 16-bit address. In fact, the value stored is the position of the sprite data (in relation to the start of the current video bank) expressed in blocks of 64 so that the value 36 in a pointer would indicate that the data for the relevant sprite is to be found at 36*64 = 2304 relative to the start of the current video bank. Sprite data, although it only requires 63 bytes, is therefore stored in 64-byte blocks for the convenience of the system, with the last byte being unused, ie the start address for a block of sprite data must be exactly divisible by 64.

2. Switching on a sprite

The fact that there is a value in one of the sprite pointers does *not* by itself mean that a sprite will appear on the screen. The VIC chip will only look for a sprite pointer and the data which it indicates if a special VIC register indicates that that sprite should be on. The [S-ENABLE] register is to be found at 53269 and it works on the principle that if bit N is set, then sprite N should be on.

3. Positioning a sprite

Each sprite has two VIC registers (and a share in a third) to define its X and Y coordinates in terms of high resolution pixels. The main registers, which we shall refer to as [SPRITE X 1], [SPRITE Y 1] and so on, are found in pairs from 53248 to 53263, with the X register being the first of each pair. Each of the two registers for a particular sprite normally define its position on the relevant coordinate between 0 and 255. Note that these positions do not correspond to normal high resolution coordinates. In the case of the vertical coordinate, the screen only contains 200 pixel positions but the sprite Y register is capable of defining 255. The actual position of the sprite is defined from the top lefthand corner of the box which contains the design and the VIC chip is quite capable of handling situations where only a part of the sprite is visible on the screen. Working from the top of the screen, the first position at which the bottom of the sprite can be seen on the screen is when the top corner is at 30 down. The last position at which the top of the sprite is visible at the bottom of the screen is 249.

When we come to the horizontal coordinate, things are not quite so simple. Like any other single-byte register, each sprite X register can record a value in the range 0–255. The screen, however, has 320 positions horizontally, quite apart from the margins added by the possibility of sprites being partly off the screen. To cope with this, another register, the 'sprite most significant bit', [SPRITE MSB], register is provided at 53264. What

this does, for each of the eight sprites, is to provide a ninth bit to be added to the end of the normal X register. With nine bits, the range which can be handled in binary is equivalent to decimal 0–511, more than enough to cope with the horizontal dimensions of the screen. For sprite N, the ninth bit will be bit N of [SPRITE MSB]. If the relevant bit is set, then the position of the sprite on the horizontal axis has 256 added to it.

4. Sprite colour

Sprite colours, which have the same numbering as normal screen colours, are controlled by eight registers between 53287 and 53294. The desired colour is simply placed into the appropriate register for the individual sprite and it changes colour.

5. Sprite size

Sprites come in normal and expanded form. Any sprite can be magnified to twice its normal size in either its horizontal or vertical component or both. In effect this means that, where one pixel would have been displayed, this is expanded to two. There is no control over the degree of magnification so all that is needed for each dimension is a register with a single bit devoted to recording whether the sprite is expanded in the particular dimension. The [EXPAND X] register is at 53277 and the [EXPAND Y] register is at 53271.

2. SPRITE PARAMETERS

Most of the sprite routines which follow use the two short routines you will enter in this section. In themselves they do nothing particularly useful but they will enable us to shorten the subsequent routines.

The first routine gets the sprite number from the BASIC instruction and checks that it is in the range 0–7 before returning it to whichever sprite command is actually being used. The second routine creates a mask which will be used by subsequent commands. No attempt is made to simulate the two routines in BASIC.

Sprite Parameters — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C400
C400               30 SYM
C400               80 GETBYT = $C006
C400   2006C0     110 GETSP JSR GETBYT
C403   C908       120 CMP #$8
C405   B001       130 BCS IQERR
C407   60         140 RTS
```

```
C408  4C48B2     150 IQERR JMP $B248
C40B  A8         180 NUMBER TAY
C40C  08         190 PHP
C40D  A901       200 LDA #1
C40F  28         210 PLP
C410  F004       220 BEQ L001
C412  0A         230 L000 ASL A
C413  88         240 DEY
C414  D0FC       250 BNE L000
C416  60         260 L001 RTS
C417             270 END

 TOTAL ERRORS IN FILE --- 0


GETBYT           C006
GETSP            C400
IQERR            C408
NUMBER           C40B
L000             C412
L001             C416

 TOTAL NUMBER OF SYMBOLS --- 6
```

Machine code

```
ADD   DATA                        CHECKSUM

C400  20 06 C0 C9 08 B0 01 60      3972
C408  4C 48 B2 A8 08 A9 01 28      5BCE
C410  F0 04 0A 88 D0 FC 60         46F8
```

Commentary

Lines 110–150: The first of the two routines mentioned above.

Lines 180–270: The second routine mentioned above. The value in A is transferred to Y. The contents of the processor status register are stored temporarily on the stack to be used later by this routine, and then the accumulator is loaded with one. The status register is recalled from the stack and a branch is made if the zero flag is set. This flag will only be set if the content of the accumulator was zero. If the sprite number was anything other than zero, the mask in the accumulator is rotated left until the set bit is in the correct position.

Testing

These routines cannot be tested at this stage.

3. SPRITE

A simple routine to switch on a particular sprite, with the added facility to define the position of the sprite when it appears. A sprite may be jumped or smoothly moved around the screen by calling this routine with a succession of different position parameters.

The procedure is:

1) Obtain the sprite number.
2) Obtain the X and Y coordinates for the sprite position.
3) Place the coordinates into the appropriate VIC registers.
4) Switch the sprite on by setting the relevant bit in the sprite control register [S-ENABLE] at 53269.

Sprite — BASIC listing

```
32000 REM*******************************
32001 REM SPRITES CONTROL MODULE
32002 REM*******************************
32010 SN=0 : SB=2^SN
32020 SX=100 : SY=100
32030 SR=53248 + 2*SN
32100 GOSUB 33000
32290 GET T$: IF T$="" THEN 32290
32999 END
33000 REM*******************************
33001 REM SPRITE ON AND POSITION
33002 REM*******************************
33040 POKE SR,SX+256*(SX>255)
33050 POKE SR+1,SY
33060 POKE 53264, (PEEK(53264)AND255-2^S
N) OR 2^SN*(SX>255)
33070 POKE 53269, PEEK (53269) OR 2^SN
33080 RETURN
```

Commentary

Line 32010: The variable SN represents the number of the sprite to be switched on. SB will hold the value of the controlling bit in [S-ENABLE].

Line 32020: The X and Y coordinates.

Line 32030: The position of the sprite X position register for the current sprite.

Lines 33040–33060: These lines place the X and Y coordinates into the appropriate registers. For a fuller explanation see the commentary on the assembly language program, lines 280–450.

Line 33070: Switch on the relevant sprite by setting a bit in [S-ENABLE].

Sprite — assembly language listing

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $C417
C417              30 SYM
C417              70 GETWRD = $C000
C417              80 GETBYT = $C006
C417              90 VIC    = $D000
C417             100 COMMA  = $AEFD
C417             110 GETSP  = $C400
C417             120 NUMBER = $C40B
C417             130 IQERR  = $C408
C417  2000C4     160 JSR GETSP
C41A  85FD       170 STA $FD
C41C  20FDAE     180 JSR COMMA
C41F  2000C0     190 JSR GETWRD
C422  A515       200 LDA $15
C424  C902       210 CMP #2
C426  B0E0       220 BCS IQERR
C428  48         230 PHA
C429  A514       240 LDA $14
C42B  48         250 PHA
C42C  20FDAE     260 JSR COMMA
C42F  2006C0     270 JSR GETBYT
C432  A5FD       280 LDA $FD
C434  0A         290 ASL A
C435  A8         300 TAY
C436  A514       310 LDA $14
C438  9901D0     320 STA VIC+1.Y
C43B  68         330 PLA
C43C  9900D0     340 STA VIC.Y
C43F  68         350 PLA
C440  AA         360 TAX
C441  A5FD       370 LDA $FD
C443  200BC4     380 JSR NUMBER
C446  85FD       390 STA $FD
C448  49FF       400 EOR #$FF
C44A  2D10D0     410 AND VIC+$10
C44D  E001       420 CPX #1
C44F  9002       430 BCC L000
C451  05FD       440 ORA $FD
C453  8D10D0     450 L000 STA VIC+$10
C456  AD15D0     460 LDA VIC+$15
C459  05FD       470 ORA $FD
C45B  8D15D0     480 STA VIC+$15
C45E  60         490 RTS
C45F             500 END
 TOTAL ERRORS IN FILE --- 0

GETWRD           C000
GETBYT           C006
```

```
VIC                 D000
COMMA               AEFD
GETSP               C400
NUMBER              C40B
IQERR               C408
L000                C453
 TOTAL NUMBER OF SYMBOLS --- 8
```

Machine code

```
ADD   DATA                              CHECKSUM

C417  20 00 C4 85 FD 20 FD AE    3BE0
C41F  20 00 C0 A5 15 C9 02 B0    36D0
C427  E0 48 A5 14 48 20 FD AE    9D48
C42F  20 06 C0 A5 FD 0A A8 A5    3DD5
C437  14 99 01 D0 68 99 00 D0    43D4
C43F  68 AA A5 FD 20 0B C4 85    8629
C447  FD 49 FF 2D 10 D0 E0 01    B8F1
C44F  90 02 05 FD 8D 10 D0 AD    5FE5
C457  15 D0 05 FD 8D 15 D0 60    55AC
C45F                             0000
```

Commentary

Lines 160–170: A call to the routine in the previous section which picks up the sprite number following the command.

Lines 190–220: Obtain the X coordinate and ensure that it falls within the range 0–511.

Lines 230–250: The X coordinate is stored temporarily on the stack.

Line 270: The Y coordinate, which must be a value in the range 0–255 and can be picked up and checked using GETBYT.

Lines 280–450: The sprite number is now picked up again from address $FD, where it was stored by line 170. It is multiplied by two by shifting left and the result is stored in the Y register. The reason for this is that the [SPRITE X] and [SPRITE Y] registers are situated at the very beginning of the VIC chip and are in pairs, one for the X coordinate and one for the Y, so the X register for a particular sprite is at an address equal to the start of the VIC chip plus twice the sprite number.

To deal with the high byte of the X coordinate, which is simply one or zero depending on whether the X coordinate is over 255, the appropriate bit needs to be set or reset in the 'most significant bit register' located at $D010. To accomplish this without affecting any of the other bits in the register, the number routine from the previous section is called up to create a mask with the correct bit set. The mask is inverted to clear the relevant bit, which is then set again if necessary.

Lines 460–490: The mask created by NUMBER is now ORed with the contents of [S-ENABLE]. The correct bit is set by this process and the sprite appears on the screen.

Testing

BASIC: The routine is self-testing. RUNning it should result in a messy sprite appearing on the screen. The reason that the sprite consists of garbage is that we have not as yet placed any sprite data into memory to define a shape. To switch off the sprite, when you are satisfied that the routine works, enter POKE 53269,0 — a more elegant method will be described in the next section. If you can see nothing at all, try altering the sprite colour as described in the machine code section below.

MACHINE CODE: The routine may be tested by means of the following line of BASIC:

```
FOR I = 0 TO 280 : SYS (50199) 0,I,200 : NEXT
```

If you have actually defined a sprite, then you should see it move across the screen. If not you should at least see a random dot pattern. There is a small possibility that the memory area from which the VIC chip is drawing the sprite data contains nothing but zeros, or that the sprite colour register in the VIC chip is defining the colour of the sprite as the same as the background colour. In either of these cases you will see nothing. Even so, if the 'nothing' goes on for around thirty seconds and is followed by the READY prompt, then it is likely that your routine is working properly. Full testing will be possible when we move on to talk about defining sprites and their colours, or you could run the line of BASIC again, first entering:

```
POKE 53287,1
```

since this will set the colour of the sprite to white. To switch the sprite off again, use the method described in the BASIC testing procedure.

Syntax and notes on use

The syntax for SPRITE is:

```
SYS (50199) { SPRITE NUMBER} ,{ X COORDINATE} ,
{ Y COORDINATE}
```

where the sprite number may be in the range 0–7, the X coordinate in the range 0–511 and the Y coordinate in the range 0–255.

It should be remembered that the possible sprite positions are not all visible on the screen, especially if you have shrunk the size of the screen.

4. DESPRITE

Since we have already have the ability to switch a sprite on, it would seem

sensible to be able to switch it off. The method is simply to set to zero the appropriate bit in the sprite control register at $D015. The command also gives the user the facility to switch off all sprites simultaneously.

Desprite — BASIC listing

```
40000 REM*******************************
40001 REM SPRITE OFF
40002 REM*******************************
40010 POKE 53269, PEEK (53269) AND (255-
2^SN)
40020 RETURN
```

Commentary

To use the routine, add this module to the two given in the last section, then add a new line to the control module:

32300 GOSUB 40000

In subsequent sections, the BASIC sprite routines will be added to this framework, with DESPRITE clearing the screen each time.

Desprite — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C4EF
C4EF               30 SYM
C4EF               90 GETSP  = $C400
C4EF              100 NUMBER = $C40B
C4EF              110 VIC    = $D000
C4EF              120 CHRGOT = $0079
C4EF  207900      150 SPROFF JSR CHRGOT
C4F2  F004        160 BEQ L000
C4F4  C93A        170 CMP #58
C4F6  D004        180 BNE L001
C4F8  A900        190 L000 LDA #0
C4FA  F00B        200 BEQ L002
C4FC  2000C4      210 L001 JSR GETSP
C4FF  200BC4      220 JSR NUMBER
C502  49FF        230 EOR #$FF
C504  2D15D0      240 AND VIC+$15
C507  8D15D0      250 L002 STA VIC+$15
C50A  60          260 RTS
C50B              270 END

 TOTAL ERRORS IN FILE --- 0

GETSP             C400
NUMBER            C40B
VIC               D000
CHRGOT            79
SPROFF            C4EF
```

```
L000              C4F8
L001              C4FC
L002              C507
 TOTAL NUMBER OF SYMBOLS --- 8
```

Machine code

```
ADD    DATA                          CHECKSUM

C4EF   20 79 00 F0 04 C9 3A D0        41C8
C4F7   04 A9 00 F0 0B 20 00 C4        3CDC
C4FF   20 0B C4 49 FF 2D 15 D0        3976
C507   8D 15 D0 60                    06BC
```

Commentary

Line 150: This is a call to the routine in the 64's ROM which picks up *again* the last character examined in the BASIC program.

Line 160: If, on return from the CHRGOT routine, the zero flag is set, then the character picked up was a byte containing zero. This indicates that the end of a line follows the call to the DESPRITE command, without any parameter defining the sprite number. The effect of the jump to a later part of the routine is to load the accumulator with zero and to load this value into the sprite control register, thus turning *all* the sprites off.

Lines 170–180: The same procedure is followed if DESPRITE is immediately followed by a colon.

Lines 190–200: These two lines load the accumulator with zero as part of the procedure for switching off all sprites. A jump is then made to the later line which stores the value held in the accumulator to [S-ENABLE].

Lines 210–220: In order for execution to reach this point in the routine, there must be a sprite number attached to the DESPRITE command. GETSP is therefore called to pick up the sprite number.

Lines 230–240: A mask is created consisting of a byte with every bit set except that corresponding to the sprite which is to be switched off. The mask is then applied to [S-ENABLE].

Line 250: The amended value is stored back into the VIC control register, switching off the specified sprite.

Testing

BASIC: The added routine is self-testing in that RUNning it should result in the sprite which is created disappearing from the screen. If the procedure

is too fast, insert a timing loop in the control module as follows:

32299 FOR I = 1 TO 2000 : NEXT I

thus creating a pause before the sprite is removed.

MACHINE CODE: The routine may be tested by switching a sprite on using the SPRITE command (see previous section), then entering:

SYS (50415),0

or

SYS (50415)

both of which should switch off the sprite.

Syntax

The syntax for DESPRITE is:

SYS (50415) [{ SPRITE NUMBER}]

where SPRITE NUMBER must be in the range 0–7. If SPRITE NUMBER is omitted (the square brackets indicate that the parameter is optional), all the current sprites will be switched off.

5. SCOLOUR

We now move on to another simple but useful command, this time to set a particular sprite to any of the 16 available colours. The colours of the eight possible sprites are controlled by eight registers beginning at 53287 in memory.

The procedure for SCOLOUR is simply:

1) Obtain the colour parameter in the range 0–15.
2) Place the value in the register corresponding to the current sprite.

SColour — BASIC listing

```
34000 REM*******************************
34001 REM SPRITE COLOUR
34002 REM*******************************
34010 POKE 53287+SN,CO
34020 RETURN
```

Commentary

The single line is self-explanatory, but to make the routine work you will need to specify the colour in the control module wth a line such as:

32040 CO = 0

and call up the SCOLOUR module with:

32200 GOSUB 34000

SColour — assembly language listing

```
ADD.  DATA      SOURCE CODE
00              10 PRT
00              20 ORG $C474
C474            30 SYM
C474            70 GETSP  = $C400
C474            80 GETBYT = $C006
C474            90 IQERR  = $C408
C474           100 VIC    = $D000
C474           110 COMMA  = $AEFD
C474  2000C4   140 SCOL JSR GETSP
C477  48       150 PHA
C478  20FDAE   160 JSR COMMA
C47B  2006C0   170 JSR GETBYT
C47E  C910     180 CMP #$10
C480  B086     190 BCS IQERR
C482  AA       200 TAX
C483  68       210 PLA
C484  A8       220 TAY
C485  8A       230 TXA
C486  9927D0   240 STA VIC+$27.Y
C489  60       250 RTS
C48A           260 END
 TOTAL ERRORS IN FILE --- 0
GETSP           C400
GETBYT          C006
IQERR           C408
VIC             D000
COMMA           AEFD
SCOL            C474
 TOTAL NUMBER OF SYMBOLS --- 6
```

Machine code

```
ADD   DATA                          CHECKSUM

C474  20 00 C4 48 20 FD AE 20        3370
C47C  06 C0 C9 10 B0 86 AA 68        5674
C484  A8 8A 99 27 D0 60              2504
```

Commentary

Lines 140–150: The sprite number is picked up by NUMBER and stored on the stack.

Lines 160–190: The colour number is picked up using GETBYT and checked to see that it is in the range 0–15.

Lines 200–240: The contents of the registers are rearranged so that the sprite number is in the Y register and the colour in the accumulator. This

enables us to use indexed addressing to place the colour value into the correct sprite colour register in the area of the VIC chip which begins at $D027 in memory.

Testing

BASIC: Run the whole of the BASIC sprite routine and you should see the sprite appear in black. Experiment with changes in the value of CO to confirm that you are correctly setting the colour.

MACHINE CODE: Load into memory the SPRITE routine you entered before this one and set sprite zero in the centre of the screen with coordinates of 160,100. Now try the following line of BASIC:

```
FOR I = 0 TO 15 : SYS (50292) 0,I : FOR J = 1 TO 1000 : NEXT J,I
```

The sprite should first of all appear black, and then turn to white and go through all the available colours. Of course, unless you have defined the sprite by some other means, all you will see is some random dots.

Syntax

The syntax for SCOLOUR is:

SYS (50292) { SPRITE NUMBER} ,{ COLOUR}

where COLOUR is a number in the range 0–15.

6. ENLARGE

Any sprite can have either its horizontal or vertical dimension stretched to twice the original size, as explained in the first section of this chapter. The following routine simplifies the whole process by allowing the user simply to specify the sprite and say which of its dimensions are to be stretched or normalised.

The procedure for ENLARGE is:

1) Obtain the sprite number.
2) Obtain the two parameters which define whether the sprite is to be enlarged along the X and Y axes.
3) Set or reset the appropriate bits in the registers at 53271 ($D017) for the Y expansion and 53277 ($D01D) for the X expansion.

Enlarge — BASIC listing

```
35000 REM*******************************
35001 REM SPRITE EXPAND
35002 REM*******************************
35010 IF XH=0 THEN POKE 53271, PEEK(5327
1) AND (255-SB)
```

```
35020 IF XV=0 THEN POKE 53277, PEEK(5327
7) AND (255-SB)
35030 IF XH=1 THEN POKE 53271, PEEK(5327
1) OR SB
35040 IF XV=1 THEN POKE 53277, PEEK(5327
7) OR SB
35050 RETURN
```

Commentary

The lines themselves are self-explanatory but you will need to include the values of XH and XV in the control module. When either of these two variables is zero,the corresponding dimension will be normal size, while if either is set to one, the corresponding dimension will be ENLARGEd. For the moment, add the following lines to the control module:

32050 XH = 1 : XV = 1
32150 GOSUB 35000

Enlarge — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C48A
C48A               30 SYM
C48A               90 GETSP  = $C400
C48A              100 GETBYT = $C006
C48A              110 COMMA  = $AEFD
C48A              120 NUMBER = $C40B
C48A              130 VIC    = $D000
C48A   2000C4     160 SIZE JSR GETSP
C48D   200BC4     170 JSR NUMBER
C490   85FD       180 STA $FD
C492   20FDAE     190 JSR COMMA
C495   2006C0     200 JSR GETBYT
C498   48         210 PHA
C499   20FDAE     220 JSR COMMA
C49C   2006C0     230 JSR GETBYT
C49F   A017       240 LDY #$17
C4A1   20A7C4     250 JSR BITSET
C4A4   68         260 PLA
C4A5   A01D       270 LDY #$1D
C4A7   AA         280 BITSET TAX
C4A8   A5FD       290 LDA $FD
C4AA   49FF       300 EOR #$FF
C4AC   3900D0     310 AND VIC.Y
C4AF   CA         320 DEX
C4B0   F005       330 BEQ L000
C4B2   CA         340 DEX
C4B3   D006       350 BNE IQERR
C4B5   05FD       360 ORA $FD
```

```
C4B7  9900D0     370 L000 STA VIC.Y
C4BA  60         380 RTS
C4BB  4C48B2     390 IQERR JMP $B248
C4BE             400 END

 TOTAL ERRORS IN FILE --- 0

GETSP            C400
GETBYT           C006
COMMA            AEFD
NUMBER           C40B
VIC              D000
SIZE             C48A
BITSET           C4A7
L000             C4B7
IQERR            C4BB
 TOTAL NUMBER OF SYMBOLS --- 9
```

Machine code

```
ADD   DATA                        CHECKSUM

C48A  20 00 C4 20 0B C4 85 FD     2FEF
C492  20 FD AE 20 06 C0 48 20     6AE0
C49A  FD AE 20 06 C0 A0 17 20     B72E
C4A2  A7 C4 68 A0 1D AA A5 FD     A157
C4AA  49 FF 39 00 D0 CA F0 05     76ED
C4B2  CA D0 06 05 FD 99 00 D0     A52C
C4BA  60 4C 48 B2                 0572
```

Commentary

Lines 160–180: The sprite number is picked up and translated into a mask. This is stored at $FD for later use.

Lines 200–210: The horizontal parameter is picked up and stored on the stack. This parameter can be either a one or a two, with one specifying normal size and two indicating double size.

Lines 220–230: The same for the vertical parameter.

Lines 240–250: These lines call up the last part of the routine as a subroutine. In this way, the latter part of the routine will be executed twice. The $17 loaded into the Y register represents the address, relative to the start of the VIC chip, of the register which controls whether each sprite is magnified vertically.

Lines 260–270: After the first call to BITSET, made by line 250, these two lines pull the parameter for the X dimension off the stack and pass it to the

routine that follows. The Y register is loaded with the address relative to the start of the VIC chip of the register which controls the horizontal magnification of each sprite.

Lines 280–370: The main part of the routine. To work, it needs to be given the parameter defining the magnification and the relative address of the register which controls either the horizontal or vertical dimension.

Lines 280–310: The parameter, one or two, is transferred to the X register. Then the relevant VIC register is taken into the accumulator unchanged, except that the bit pointing to the specified sprite is cleared, regardless of its original condition, by use of the mask created by NUMBER.

Lines 320–330: The X register is tested to see if the magnification factor is one. If so the value in the accumulator is stored in the VIC register.

Lines 340–350: The content of the X register is tested to see whether it is two. If not an error message is generated.

Lines 360–370: At this point, the content of the X register must be one. Accordingly the mask is used to set the relevant bit and the result is stored back in the VIC register.

Testing

BASIC: Running the whole BASIC routine should result in an expanded version of the same garbled sprite being seen. We can only really see how effective the expansion is when we have given ourselves the ability to define sprite shapes. You might like to play with the values of XV and XH to confirm that either dimension can be expanded or shrunk to normal size.

MACHINE CODE: First make a sprite visible on the screen by using the SPRITE and SCOLOUR commands to place it somewhere in the middle of the screen and in a clearly contrasting colour. Now enter:

```
SYS (50314) 0,2,2
```

and the sprite should expand in both directions. Enter:

```
SYS (50314) 0,1,2
```

and the sprite should contract along the horizontal axis.
Finally try:

```
SYS (50314) 0,1,1
```

and it should return to its original size.

Syntax

The syntax for ENLARGE is:

SYS (50314) { SPRITE NUMBER} ,{ X MAGNIFICATION FACTOR} ,{ Y MAGNIFICATION FACTOR}

The magnification factor can be either one or two with two representing a doubling of length along the specified dimension and one restoring the normal size.

7. SPTR

As we have already seen in the introduction to sprites, the eight sprites which the VIC chip is capable of handling at one and the same time do not always have to be the *same* eight sprites. If the pointer for a sprite is changed so that it indicates a different set of data, that sprite will change as the VIC chip picks up the new definition from memory, thus allowing a single sprite to assume a series of shapes, perhaps for the purposes of animation.

The procedure for SPTR is as follows:

1) Obtain the sprite number.
2) Calculate the position of the sprite pointers in memory using the [EDITOR] register.
3) Obtain the address at which the sprite data is to be found.
4) Translate the sprite data address into the number of a 64-byte block within the current video bank.
5) Store the block number in the appropriate sprite pointer.

SPTR — BASIC listing

```
36000 REM*****************************
36001 REM SPRITE POINTER
36002 REM*****************************
36010 SA=PEEK(648)*256
36020 SP=SA+1016
36030 POKE SP+SN,SD
36100 RETURN
```

Commentary

Lines 36010–36020: The address of the first sprite pointer is calculated.

Line 36030: The sprite data block number (SD) is POKEd into the relevant sprite pointer.

To run the routine, you will need to define the sprite data block in the control module and call up the SPTR module. Add the following lines to

the control module.

32210 SD = 13
32220 GOSUB 36000

See the section on testing for the effect of these lines.

SPTR — assembly language listing

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $C4BE
C4BE              30 SYM
C4BE             100 GETSP  = $C400
C4BE             110 GETWRD = $C000
C4BE             130 COMMA  = $AEFD
C4BE  2000C4     150 SPTR JSR GETSP
C4C1  48         160 PHA
C4C2  20FDAE     170 JSR COMMA
C4C5  AD8802     180 LDA 648
C4C8  18         190 CLC
C4C9  6903       200 ADC #3
C4CB  85FE       210 STA $FE
C4CD  A9F8       220 LDA #$F8
C4CF  85FD       230 STA $FD
C4D1  2000C0     240 JSR GETWRD
C4D4  A515       250 LDA $15
C4D6  4D8802     260 EOR 648
C4D9  29C0       270 AND #$C0
C4DB  D00F       280 BNE BANKER
C4DD  0614       290 ASL $14
C4DF  2615       300 ROL $15
C4E1  0614       310 ASL $14
C4E3  2615       320 ROL $15
C4E5  68         330 PLA
C4E6  A8         340 TAY
C4E7  A515       350 LDA $15
C4E9  91FD       360 STA ($FD).Y
C4EB  60         370 RTS
C4EC  4C9AC0     380 BANKER JMP $C09A
C4EF             390 END
 TOTAL ERRORS IN FILE --- 0


GETSP            C400
GETWRD           C000
COMMA            AEFD
SPTR             C4BE
BANKER           C4EC
 TOTAL NUMBER OF SYMBOLS --- 5
```

Machine code

```
ADD    DATA                              CHECKSUM
C4BE   20 00 C4 48 20 FD AE AD    33FD
C4C6   88 02 18 69 03 85 FE A9    52E1
C4CE   F8 85 FD 20 00 C0 A5 15    C33F
C4D6   4D 88 02 29 C0 D0 0F 06    54B4
C4DE   14 26 15 06 14 26 15 68    184A
C4E6   A8 A5 15 91 FD 60 4C 9A    938A
C4EE   C0                         00C0
```

Commentary

Lines 150–160: The sprite pointer specified is picked up by the GETSP routine and then saved on the stack.

Lines 180–230: These lines calculate the start address of the sprite pointers and store it in the two-byte address at $FD. The method is first to pick up the contents of location 648, which is the system's pointer to the location of the screen, and then to add 1016, thus creating a two-byte value representing the position of the first sprite pointer in memory.

Lines 240–280: The address of the sprite definition is picked up using GETWRD and checked to see that it is in the correct video bank.

Lines 290–320: The sprite pointer does not specify a complete address but the number of a 64-byte block. These lines perform the necessary transformation to that format.

Lines 330–360: The sprite number is recalled from the stack and placed into the Y register so that it can be used for indexed addressing. The correct 64-byte block is already stored in $15 and the start address of the sprite pointers in the two-byte pointer at $FD. All that needs to be done, therefore, is to store the value from $15 into the address specified by the pointer at $FD plus the sprite number.

Line 380: A jump to the WRONG BANK error message, if needed.

Testing

BASIC: Enter lines 10–30 of the machine code testing program below *but* number them 32230–32250. The effect should be the same as for the machine code routine.

MACHINE CODE: The following short BASIC program defines a sprite of three upright bars. First switch on sprite zero using the routines already entered, then enter the following program:

```
 5 SYS (50366) 0,8192
10 FOR I = 832 TO 832 + 62
20 POKE I,56
30 NEXT I
```

The sprite pointer is set first so that you can see the redefinition take place on the screen. The sprite data is stored in the cassette buffer, a useful temporary storage space for all kinds of data.

Syntax and notes on use

The syntax for SPTR is:

SYS (50366) { SPRITE NUMBER} ,{ ADDRESS}

The sprite number must be in the range 0–7, and the address must be within the current 16K video bank. Note also that to perform sensibly, the address provided for the start of the sprite data must be a multiple of 64, since the sprite pointers themselves can only work in blocks of 64 bytes. One easy way to ensure that an error is not made is to enter the sprite data address in the format BLOCK NUMBER*64, eg 13*64 = 832.

8. RESPRITE

This command, whose value cannot be properly assessed until the next section has also been entered, handles a separate DATA pointer intended to simplify the definition of sprites within a program. In normal BASIC, the READ statement makes use of a pointer which moves through the program, recording which item of DATA is next to be processed by a READ statement. In the companion book to this one, *Commodore 64 Machine Code Master*, we showed how the RESTORE command, which normally resets the data pointer to the beginning of a program, could be redefined so that the user could specify which *line* in the program would be indicated, with the first item of DATA to be read being the first item on or after that line. The advantage of this is that the program can be enabled to READ specific sections of DATA taken from different parts of the program.

The current command gives you the same kind of facility, but this time a new pointer is involved which will be used to locate special strings which will allow sprites to be defined graphically within a program rather than in the form of numbers.

The procedure for RESPRITE is as follows:

1) Obtain the line number to which the 'restore' is to be made — if no line number is specified then the pointer will be moved to the start of the program.
2) Use an existing ROM routine to find the start address of the specified line in the memory.

3) Store the result in the RESPRITE pointer.

It is not feasible to duplicate this particular command in BASIC. Those readers wishing to use DATA statements to store sprite data, in accordance with the BASIC method below, are referred to David Lawrence's *Advanced Programming Techniques on the Commodore 64* for some suggestions as to how DATA statements can be more flexibly handled.

Resprite — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C50B
C50B               30 SYM
C50B              100 SPRPTR = $02E9
C50B              110 FINDLN = $A613
C50B              120 GETWRD = $C000
C50B              130 CHRGOT = $0079
C50B   207900     160 SRES  JSR CHRGOT
C50E   F013       170 BEQ L000
C510   C93A       180 CMP #58
C512   F00F       190 BEQ L000
C514   2000C0     210 JSR GETWRD
C517   2013A6     220 JSR FINDLN
C51A   A55F       230 LDA $5F
C51C   A660       240 LDX $60
C51E   B007       250 BCS L001
C520   4CE3A8     270 JMP $A8E3
C523   A52B       290 L000 LDA $2B
C525   A62C       300 LDX $2C
C527   38         320 L001 SEC
C528   E901       330 SBC #1
C52A   8DE902     340 STA SPRPTR
C52D   B001       350 BCS L002
C52F   CA         360 DEX
C530   8EEA02     370 L002 STX SPRPTR+1
C533   60         380 RTS
C534               390 END

 TOTAL ERRORS IN FILE --- 0


SPRPTR             2E9
FINDLN             A613
GETWRD             C000
CHRGOT             79
SRES               C50B
L000               C523
L001               C527
L002               C530
 TOTAL NUMBER OF SYMBOLS --- 8
```

Machine code

```
ADD   DATA                          CHECKSUM

C50B  20 79 00 F0 13 C9 3A F0        4260
C513  0F 20 00 C0 20 13 A6 A5        1EBD
C51B  5F A6 60 B0 07 4C E3 A8        73D6
C523  A5 2B A6 2C 38 E9 01 8D        7AB3
C52B  E9 02 B0 01 CA 8E EA 02        956E
C533  60                             0060
```

Commentary

Lines 160–190: A test to see whether the command is immediately followed by the end of the line or a colon. If so, the pointer will be restored to the beginning of the program.

Lines 210–220: To have reached this point, there must be a parameter following the command, specifying the line number to which the pointer will be restored. These lines pick up the number and then call up the ROM subroutine used by BASIC to find the start address of a particular line in the program.

Lines 230–270: The address returned is placed into the accumulator and the X register. A test is then made of the carry flag. The reason for this is that the ROM subroutine called by line 220 sets the carry flag if it finds the line specified in the program. If the line specified was *not* found in the program, a jump is made to the UNDEFINED LINE error message.

Lines 290–300: This is the line to which execution is sent if there was no line parameter given. To effect a restoration to the beginning of the program, the start-of-BASIC pointer is picked up and placed into the accumulator and the X register.

Lines 320–370: For reasons that will be explained in the commentary on the next command, the actual sprite defining routine will require the pointer value to have been reduced by one. This is done, with provision for a carry, and the result stored in the two-byte location at SPRPTR, two bytes which happen to be unused by BASIC.

Testing

Since we do not yet have any means of defining the special sprite data, it is not possible to make a very meaningful test of the command at this point. Entering the following lines of BASIC in direct mode should, however, show up any problems which might have occurred.

POKE 745,0 : POKE 746,0
SYS (50443)
PRINT PEEK (745), PEEK (746)

Provided that the start-of-BASIC pointer has not been changed (by ALLOT for instance), the results returned should be:

0 8

All that we have done here is to test that the routine is capable of restoring the pointer to the beginning of the BASIC program.

Syntax and notes on use

The syntax RESPRITE is:

SYS (50443) [{ LINE NUMBER}]

where LINE NUMBER, if it is included, must refer to an existing line within the program. If no line number is included the sprite data pointer is set to the beginning of the program.

9. SHAPE

Anyone who has worked with sprites will be aware that, though defining a sprite is not exactly difficult, it is certainly messy. With the command in this section, SHAPE, all the problems are over. From now on, you will be able to define a sprite from within a BASIC program in a way that makes absolutely clear what the eventual result will be and without any need for calculation and POKEing of numbers into memory.

The procedure for SHAPE is as follows:

1) Obtain the address at which the sprite data is to be stored in memory, ie the start of 64-byte block.
2) Pick up a string defining a single horizontal line across a sprite.
3) Translate that string, eight characters at a time, into three bytes, with each position in the eight-character string representing one bit. A bit will be set if the corresponding character in the string is '*' and reset otherwise.
4) Store each byte, as it is created, in the specified 64-byte area of memory, one after another.
5) When a single 24-character string has been processed pick up the next until 21 have been dealt with. If a full 21 strings are not found, generate an error message.

Shape — BASIC listing

```
41000 REM********************************
41001 REM SPRITE DEFINE
```

```
41002 REM*******************************
41010 FOR I=0 TO 20
41020 READ SP$
41100 FOR J=1 TO 17 STEP 8
41110 BYTE=0
41200 FOR K=0 TO 7
41210 BYTE=BYTE-2^(7-K)*(MID$(SP$,J+K,1)
="*")
41250 NEXT K
41270 POKE 64*SD+I*3+(J-1)/8,BYTE
41300 NEXT J
41400 NEXT I
41999 RETURN
42000 REM*******************************
42001 REM SPRITE DATA
42002 REM*******************************
42010 DATA ********         ********
42020 DATA ********         ********
42030 DATA ********         ********
42040 DATA ************************
42050 DATA ********         ********
42060 DATA ********         ********
42070 DATA ********         ********
42080 DATA ************************
42090 DATA ************************
42100 DATA ********         ********
42110 DATA ********         ********
42120 DATA ********         ********
42130 DATA ************************
42140 DATA ************************
42150 DATA ************************
42160 DATA ********         ********
42170 DATA ********         ********
42180 DATA ********         ********
42190 DATA ********         ********
42200 DATA ********         ********
42210 DATA ********         ********
```

Commentary

Before entering the module, enter a new line as follows:

32260 GOSUB 41000

Lines 41010–41020: This loop ensures that a full 21 lines of the sprite are picked up.

Lines 41100–41110: The string, which should be 24 characters long, will be processed in three groups of eight characters, ie 1–8, 9–16, 17–24. The variable BYTE will be used to store the value which will result from the translation of each eight-character group.

Lines 41200–41250: Each group of characters is translated into a value in such a way that an asterisk will represent a set bit in an eight-bit value. Note that the string is scanned from left to right, so the number of the bit within the byte has to be a mirror image of the position of the character in the group.

Line 41270: The number of the 64-byte block in which the sprite data is to be stored is represented by SD.

Lines 42010–42210: These lines illustrate the method involved in defining the sprite. Each data line is 24 characters long and there are precisely 21. Note that if you wish to use a different design, lines ending in spaces will have to be enclosed in quotation marks.

Shape — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $C534
C534               30 SYM
C534              110 LENGTH = $006A
C534              120 PTR    = $006B
C534              130 LETTER = $B113
C534              140 SPRPTR = $02E9
C534              150 IP     = $007A
C534              160 GETWRD = $C000
C534              170 GETSTR = $C089
C534              180 LETTOK = $0088
C534   207300      230 GETCHR JSR $0073
C537   D013        240 BNE L001
C539   A001        250 LDY #1
C53B   B17A        260 LDA (IP).Y
C53D   F00E        270 BEQ L002
C53F   18          290 CLC
C540   A57A        300 LDA IP
C542   6904        310 ADC #4
C544   857A        320 STA IP
C546   9002        330 BCC L000
C548   E67B        340 INC IP+1
C54A   A93A        350 L000 LDA #58
C54C   60          360 L001 RTS
C54D               380 L002
C54D   4CA1C0      390 JMP $C0A1
C550               420 GETLN
C550   2034C5      430 JSR GETCHR
C553   C93A        440 CMP #58
C555   F0F9        450 BEQ GETLN
C557   C988        460 CMP #LETTOK
```

```
C559  F0F5       470 BEQ GETLN
C55B  C953       480 CMP #83
C55D  F009       490 BEQ L004
C55F             510 L003
C55F  2034C5     520 JSR GETCHR
C562  C93A       530 CMP #58
C564  D0F9       540 BNE L003
C566  F0E8       550 BEQ GETLN
C568             560 L004
C568  2034C5     570 JSR GETCHR
C56B  C950       580 CMP #80
C56D  D0F0       590 BNE L003
C56F             600 L005
C56F  2034C5     610 JSR GETCHR
C572  90FB       630 BCC L005
C574  2013B1     640 JSR LETTER
C577  B0F6       650 BCS L005
C579  C924       660 CMP #36
C57B  D0EB       670 BNE L004
C57D  2034C5     690 JSR GETCHR
C580  C9B2       700 CMP #$B2
C582  D00F       720 BNE SNERR
C584  207300     730 JSR $0073
C587  2089C0     740 JSR GETSTR
C58A  A56A       760 LDA LENGTH
C58C  C918       770 CMP #24
C58E  F006       780 BEQ CONVER
C590             800 IQERR
C590  4C48B2     810 JMP $B248
C593             830 SNERR
C593  4C08AF     840 JMP $AF08
C596             860 CONVER
C596  209CC5     870 JSR CON1
C599  209CC5     880 JSR CON1
C59C             900 CON1
C59C  A007       910 LDY #7
C59E             920 L006
C59E  B16B       930 LDA (PTR).Y
C5A0  C92A       940 CMP #42
C5A2  F002       950 BEQ L007
C5A4  18         960 CLC
C5A5  24         970 BYT $24
C5A6             980 L007
C5A6  38         990 SEC
C5A7  6614      1000 ROR $14
C5A9  88        1010 DEY
C5AA  10F2      1020 BPL L006
C5AC  A56B      1040 LDA PTR
C5AE  18        1050 CLC
C5AF  6908      1060 ADC #8
C5B1  856B      1070 STA PTR
```

```
C5B3  9002      1080 BCC L008
C5B5  E66C      1090 INC PTR+1
C5B7            1100 L008
C5B7  C8        1120 INY
C5B8  A514      1130 LDA $14
C5BA  91FD      1140 STA ($FD).Y
C5BC  E6FD      1160 INC $FD
C5BE  D002      1170 BNE L009
C5C0  E6FE      1180 INC $FE
C5C2            1190 L009
C5C2  60        1200 RTS
C5C3            1230 SDEF
C5C3  2000C0    1250 JSR GETWRD
C5C6  A514      1260 LDA $14
C5C8  A615      1270 LDX $15
C5CA  85FD      1280 STA $FD
C5CC  86FE      1290 STX $FE
C5CE  A57A      1310 LDA IP
C5D0  48        1320 PHA
C5D1  A57B      1330 LDA IP+1
C5D3  48        1340 PHA
C5D4  ADE902    1360 LDA SPRPTR
C5D7  AEEA02    1370 LDX SPRPTR+1
C5DA  857A      1380 STA IP
C5DC  867B      1390 STX IP+1
C5DE  A215      1410 LDX #21
C5E0            1420 L010
C5E0  8A        1430 TXA
C5E1  48        1440 PHA
C5E2  2050C5    1450 JSR GETLN
C5E5  68        1460 PLA
C5E6  AA        1465 TAX
C5E7  A57A      1470 LDA IP
C5E9  D002      1480 BNE L011
C5EB  C67B      1490 DEC IP+1
C5ED            1500 L011
C5ED  C67A      1510 DEC IP
C5EF  CA        1520 DEX
C5F0  D0EE      1530 BNE L010
C5F2  A57A      1550 LDA IP
C5F4  A67B      1560 LDX IP+1
C5F6  8DE902    1570 STA SPRPTR
C5F9  8EEA02    1580 STX SPRPTR+1
C5FC  68        1600 PLA
C5FD  857B      1610 STA IP+1
C5FF  68        1620 PLA
C600  857A      1630 STA IP
C602  60        1640 RTS
C603            1650 END

 TOTAL ERRORS IN FILE --- 0
```

```
LENGTH              6A
PTR                 6B
LETTER              B113
SPRPTR              2E9
IP                  7A
GETWRD              C000
GETSTR              C089
LETTOK              88
GETCHR              C534
L000                C54A
L001                C54C
L002                C54D
GETLN               C550
L003                C55F
L004                C568
L005                C56F
IQERR               C590
SNERR               C593
CONVER              C596
CON1                C59C
L006                C59E
L007                C5A6
L008                C5B7
L009                C5C2
SDEF                C5C3
L010                C5E0
L011                C5ED
 TOTAL NUMBER OF SYMBOLS --- 27
```

Machine code

```
ADD   DATA                              CHECKSUM

C534  20 73 00 D0 13 A0 01 B1    3D8B
C53C  7A F0 0E 18 A5 7A 69 04    8426
C544  85 7A 90 02 E6 7B A9 3A    7DC8
C54C  60 4C A1 C0 20 34 C5 C9    6743
C554  3A F0 F9 C9 88 F0 F5 C9    8F63
C55C  53 F0 09 20 34 C5 C9 3A    6F20
C564  D0 F9 F0 E8 20 34 C5 C9    D6E3
C56C  50 D0 F0 20 34 C5 90 FB    82CF
C574  20 13 B1 B0 F6 C9 24 D0    41CC
C57C  EB 20 34 C5 C9 B2 D0 0F    9B0F
C584  20 73 00 20 89 C0 A5 6A    37BC
C58C  C9 18 F0 06 4C 48 B2 4C    8E10
C594  08 AF 20 9C C5 20 9C C5    4625
C59C  A0 07 B1 6B C9 2A F0 02    7762
C5A4  18 24 38 66 14 88 10 F2    2632
C5AC  A5 6B 18 69 08 85 6B 90    7A8A
C5B4  02 E6 6C C8 A5 14 91 FD    5C17
C5BC  E6 FD D0 02 E6 FE 60 20    D868
C5C4  00 C0 A5 14 A6 15 85 FD    4D6B
```

```
C5CC  86 FE A5 7A 48 A5 7B 48   A4D2
C5D4  AD E9 02 AE EA 02 85 7A   A4BC
C5DC  86 7B A2 15 8A 48 20 50   7D50
C5E4  C5 68 AA A5 7A D0 02 C6   A3EA
C5EC  7B C6 7A CA D0 EE A5 7A   96DC
C5F4  A6 7B 8D E9 02 8E EA 02   960E
C5FC  68 85 7B 68 85 7A 60      38F8
```

Commentary

Lines 230–360: A subroutine employed by the main part of the routine. Its overall function is to pick up the first character in the BASIC program following the address indicated by the sprite data pointer, to skip over the link bytes and line number bytes which begin a line in BASIC and to test whether the sprite data pointer has moved past the end of the program. If it has, then a new error message OUT OF SPRITE DATA is generated.

Line 230: CHRGET is used to pick up the next character after the position indicated by the sprite data pointer.

Line 240: If the byte picked up is not a zero, representing the end of the line, execution of the subroutine ceases. This ensures that the main routine begins its work at the correct place, which is the beginning of a line.

Lines 250–270: A check for the end of the file. This is done by examining the next byte along, which should be the high byte of the two link bytes which precede a line in BASIC. If the byte picked up is zero then the end of the program has been reached and a branch is made to the OUT OF SPRITE DATA error message.

Lines 300–340: To have reached this point means that the end of file has not been encountered. Four is added to the value of the 'interpretative pointer' (which currently holds the position within the BASIC program) thus jumping over the two link bytes and the two line number bytes to the beginning of the line of BASIC text.

Line 350: At a later point in the routine you will see that no distinction is made between the procedure to be followed after a line start and a colon. To simplify matters, this line places the value 58 into the accumulator to signify that a colon has been found.

Line 360: The line used to exit the routine.

Line 390: A jump to the OUT OF SPRITE DATA error message.

Lines 420–780: These lines, which will be commented on in greater detail, find the next occurrence of a special kind of string called SP$, which holds sprite data.

Lines 430–470: If there is a LET or a colon at the beginning of the BASIC line (it is quite legal to start a line with a colon), they are skipped over.

Lines 480–490: The format of an SP$ statement always has 'SP' at the beginning (allowing for LET or a colon) so these lines scan the next character in the BASIC program to see if it is an 'S'. If it is, then a jump is made to the next test.

Lines 510–540: If the first character was not an S, the BASIC line is scanned until a colon is found. To do this, the earlier routine GETCHR is used, beginning at line 230, to pick up and test each character in turn. You can see now why line 350 loads the accumulator with the ASCII value of a colon. The value 58 in the accumulator acts as a simple signal to begin searching for an SP$, whether it is the start of a new line or merely of a new statement following a colon.

Line 550: If the value of a colon is returned in the accumulator, a branch is made back to the beginning of the section which deals with line starts.

Lines 560–590: These lines, which are only executed when the first effective character of the BASIC line or statement is an S, test that the next character is a 'P'. If not, lines 510–540 are called up again to seek out the next line start or colon.

Lines 610–650: As mentioned earlier, a SP$ statement always begins with 'SP' as in 'SP$ = '. What comes in between the 'SP' and the '$' is a matter of indifference — as with all variable names on the 64, only the first two characters matter. These lines skip over the rest of the string name, if any, by ignoring any letter or digit following the 'SP'.

Lines 660–670: The first character which is not a letter or number is tested to see if it is a '$' sign. If not, execution jumps back to the part of the routine which searches for the next SP$.

Lines 700–720: If the string symbol was present, the next check is that it is followed by an equals sign. If not, back to the search.

Lines 730–740: The GESTSTR routine is used to pick up the string following the equals sign. The flexibility of GETSTR means that string expressions may be used if desired.

Lines 770–810: These lines check that the length of the SP$ is 24 characters.

Line 840: This line is called from line 720 if the format of the SP$ statement is wrong and is a simple jump to the SYNTAX ERROR message.

Lines 860–1200: This is the heart of the SHAPE routine, its purpose being to take information from the special SP$ data and transform it into sprite data in the memory.

Lines 870–880: There are 24 characters in each string being transformed into sprite data and these must be stored in three bytes — the routine to translate a series of eight characters into a single byte will therefore be executed three times for each of the 21 lines of the sprite. These two lines call the conversion section, which follows them immediately, twice. Once the two lines have been executed, the conversion routine will naturally be executed a third time simply because it is the next section of code.

Line 910: The Y register is initialised with the value seven, indicating the first bit to be translated from the string of characters.

Line 930: The accumulator is loaded with the character currently pointed to by the sprite data pointer.

Lines 940–990: These lines set or clear the carry flag according to whether or not the character picked up from the SP$ was an asterisk. It is interesting to note the use of the BYT directive here, which inserts a single byte with the value of $24 in to the machine code at this point. The effect of this byte, unless it is jumped over, is to render the following SEC instruction inoperative by making it temporarily part of a two-byte BIT instruction. The BIT instruction is irrelevant to the execution of the routine, but it effectively wipes out the SEC without having to make a jump over it, thus saving two bytes.

Line 1000: The contents of the carry flag, which now indicates whether an asterisk was picked up from the SP$ or not, is rotated into the byte at $14, which is being used to assemble the current byte of sprite data.

Lines 1010–1020: These lines test whether the loop beginning at line 910 has been executed eight times.

Lines 1040–1080: Having dealt with eight characters, the pointer which records the position of the beginning of the string is advanced by eight.

Line 1120: At this point, to have failed the test in line 1020, the contents of

the Y register must have reduced below zero, or rather reset to 255. This line re-initialises the Y register to zero.

Lines 1130–1180: The byte of sprite data just created is placed in the memory address indicated by the sprite byte pointer which is stored at $FD–FE. The sprite byte pointer is incremented by one.

Lines 1220–1640: After all the subroutines, we finally come to the main part of the sprite routine, from which the subroutines are called. It is important to remember in studying the lines which follow, that they are what happens *first* when this command is used.

Lines 1250–1290: As a first step, the address at which the sprite data is to be placed in memory is picked up using GETWRD and stored in the 'sprite byte pointer' at $FD–FE.

Lines 1310–1340: The two-byte address currently pointed to by the interpretative pointer, a system pointer which records the current position in the BASIC program, is stored on the stack.

Lines 1360–1390: The contents of the sprite data pointer are loaded into the interpretative pointer so that we can use ROM routines to perform much of the subsequent work.

Lines 1410–1440: This is the beginning of the main loop for the routine. The X register is loaded with 21, the number of lines in a sprite, and this value is saved on the stack.

Line 1450: The GETLN subroutine, which begins at line 420, is called to find the first occurrence of an SP$ and to transfer the data to a proper sprite definition at the correct location in memory.

Lines 1460–1465: The value held in the X register before the call to the subroutines is recovered, since it ordinarily held the number of lines of the sprite which still had to be defined.

Lines 1470–1510: The interpretative pointer is now backed up by one place, since the last call to the GETLN is one character past the beginning of the line following.

Lines 1520–1530: The X register is decremented and another line of sprite data picked up if the contents have not yet reached zero.

Lines 1550–1580: The address currently stored in the interpretative

pointer for immediate use is transferred to the sprite data pointer, so that that pointer's position is permanently updated before the return to BASIC.

Lines 1600–1640: The original value of the interpretative pointer is restored in order that BASIC execution may continue normally.

Testing

BASIC: When run, you should see a sprite defined, consisting of a ladder whose rungs become thinner as they ascend. Note that the speed with which the sprite appears is the speed at which it is *defined*. If the same sprite is subsequently switched on without being redefined, it will appear instantaneously. Note also that the sprite does not have to be visible while it is being defined.

MACHINE CODE: In order for this command to be tested there must first be a BASIC program containing a proper definition of a sprite. The following short program is designed to illustrate how sprite definition is achieved and needs all the sprite routines given so far, resident in the memory.

```
10 REM ROUTINE TO TEST SPRITE DEFINER RO
UTINE
15 SYS (50443) 50
20 SYS (50292) 0,1 : REM SPRITE 0 TO WHI
TE
30 SYS (50199) 0,200,200 : REM SPRITE 0
ON
40 SYS (50366) 0,832 : REM SET SPRITE 0
POINTER TO START OF CASS. BUFFER
50 REM SPRITE DATA
60 SP$  = "**************************"
70 SP$  = "**************************"
80 SP$  = "********          ********"
90 SP$  = "********          ********"
100 SP$ = "********          ********"
110 SP$ = "********          ********"
120 SP$ = "********          ********"
130 SP$ = "********          ********"
140 SP$ = "********          ********"
150 SP$ = "********          ********"
160 SP$ = "********          ********"
170 SP$ = "********          ********"
180 SP$ = "********          ********"
190 SP$ = "********          ********"
200 SP$ = "********          ********"
210 SP$ = "********          ********"
220 SP$ = "********          ********"
```

```
230 SP$ = "********          ********"
240 SP$ = "********          ********"
250 SP$ = "************************"
260 SP$ = "************************"
300 SYS (50627) 832
```

The program, when RUN, will do exactly what the BASIC test program in the SPTR section did, but a glance at the program shows that, instead of meaningless numbers, the shape of the sprite is clearly defined in the program lines.

Syntax and notes on use

The syntax for SHAPE is:

SYS (50627) { ADDRESS}

ADDRESS may refer to any address in the memory of the 64 but remember that, if it is not divisible exactly by 64 or if it falls outside the current video block, the system will not be able to make use of it. If you are in doubt as to whether the address being specified is that of a 64-byte block, a simple answer would be to replace the straight address with an expression, as in:

SYS (50627) 13*64

which is exactly the same as specifying a start address of 832. The sprite data on which the command works must be in the form:

LET SP$ = { STRING EXPRESSION}

The LET and the spaces between the elements of the statement are optional, but the string must have a length of 24 characters and only positions in the string which are filled with an asterisk (*) will be translated into set pixels in the eventual sprite.

Each time the SHAPE command is used, there must be a full 21 SP$ sprite data statements somewhere in the program for it to call on. The program can contain more than 21 sprite data statements. Using RESPRITE, it is possible to move the sprite data pointer so that SHAPE begins picking up sprite data at any point in the program. If, however, the SHAPE command is used and there are less than 21 SP$ sprite data statements between the current position of the sprite data pointer and the end of the program, the program will stop with an OUT OF SPRITE DATA error message.

Unlike the DATA pointer, the sprite data pointer is *not* set to the beginning of the program whenever RUN is used, so RESPRITE must be used before beginning the process of defining sprites.

You might like to note that, with a little alteration, this routine could be employed as a user-defined character generator. Since a normal character

is only 8*8 pixels rather than 24*21, you would have to alter line 1410 to read:

1410 LDX #8

and line 1450 to read:

1450 JSR CON1

Once the routines to move the character data into user RAM have been carried out, this routine could be used to pick up the definition of a character and place it into character memory at a specified point — we leave it to you to sort out the problems.

CHAPTER 5
Sound Commands

1. SOUND IN GENERAL

In the sections which follow, you will find that the approach followed in the book changes substantially. While for the first two sound commands, BEEP and CHORD, we have provided BASIC simulations, in the later sections we leave the comfort of BASIC entirely. The reason for this is partly tied up with the nature of the Sound Interface Device (SID) itself. Unlike the VIC chip, the SID is not an integral part of the 64's system. Where the VIC chip is constantly performing important tasks within the system and communicating with the rest of the 64, the SID stands relatively aloof. It is a separate piece of equipment which deigns to take instructions from the 64 but is not really part of it.

The SID chip communicates with the rest of the system, like the VIC chip, by means of registers, but even here its separateness is emphasised by the fact that, although the CPU can place data *into* those registers, the information cannot be read *out* of them — they are a special kind of memory known as 'write only'. In addition, there are no simple commands to the SID. Any single note which the SID sounds has to be based on a whole complex of values which have first of all been provided for it. Once the instruction to sound a note is given, the SID will continue the development of the note through its various stages quite independently of what is happening elsewhere within the 64. For that reason, any sets of commands which are to work together in driving the SID are quite distinct from the other commands in BASIC. The vast majority of them will achieve absolutely nothing on their own, they merely contribute to the nature of the sound which the SID will ultimately produce. We found in practice that to reproduce these commands in BASIC required more extensive space being given over to the BASIC routines than for the other commands, and produced effects which could not be tested until a complex mass of interrelated routines had been entered. The final command, PLAY, which alone justifies the effort of entering the routines which precede it, is a very complex and lengthy piece of machine code and required extensive BASIC programming to simulate.

The commands you will find given here in no way exhaust the capabilities of the SID, because those capabilities are quite enormous in terms of

their range and flexibility. Wide as its capabilities may be, however, there is relatively little technical information on the nature of the SID. We do not intend, therefore, to attempt a general description of the chip's many capabilities but to allow the descriptions of the individual commands to introduce you to what can be achieved. For anyone who wants to know more, the best source is the Commodore 64 Programmer's Reference Guide.

Having said all this, the final command in this chapter, PLAY, will introduce you to probably as wide a range of the SID's capabilities as you will ever be able to employ practically, to make regular use of music in your programs a real possibility for the first time.

2. BEEP

Simply because the SID is a sophisticated piece of equipment with a range of capabilities far beyond everyday programming needs, we must not be carried away with the idea that every use of the chip must be complex. In probably the majority of cases within the program, a sound is used either as a reminder that some process has been completed which takes a considerable time and during which the user may lose concentration, or as an indication that an instruction of some kind is expected. The BEEP command does just that, allowing the user to specify a single note of specified duration and frequency without the need to attach complex parameters. Used in conjunction with a loop in BASIC and reading its parameters from DATA statements, BEEP can be used to play simple tunes without the need to set up the complex SID registers.

The procedure for BEEP is as follows:

1) Turn off all the three sound channels or 'voices' in the SID.
2) Turn the SID volume to full and switch off all its filters (see later for an explanation of the SID's sound filtering capabilities).
3) Set up the characteristics of the note to be played, or ADSR (again see later), so that a flat tone will be sounded.
4) Obtain the length of the note to be played.
5) Obtain the frequency of the note to be played.
6) Switch on the SID, specifying that the note be played using a sawtooth waveform.
7) Wait for the specified duration of the note.
8) Switch off the SID.

Beep — BASIC listing

```
45000 REM*******************************
45001 REM SOUND CONTROL MODULE
45002 REM*******************************
45010 SID=54272
```

```
45020 FOR I=4 TO 18 STEP 7
45030 POKE SID+I,0
45040 NEXT I
45100 NL=500
45110 FR=8000
45200 GOSUB 46000
45999 END
46000 REM*******************************
46001 REM BEEP
46002 REM*******************************
46010 POKE SID+24,15
46020 POKE SID+23,8
46030 POKE SID+5,0
46040 POKE SID+6,240
46050 HF=INT(FR/256)
46060 POKE SID,FR-256*HF
46070 POKE SID+1,HF
46080 POKE SID+4,17
46100 FOR I=1 TO NL : NEXT I
46200 POKE SID+4,0
46999 RETURN
```

Commentary

Line 45010: The start address of the SID chip is stored in the variable SID.

Lines 45020–45040: The SID has three separate voices which are largely controlled by three separate sets of seven registers. The register which controls whether a voice is on or not is the fifth of each set of seven. This loop simply ensures that every voice is switched off.

Lines 45100–45110: The note length is stored in NL and the 'frequency' in FR. Note that this second value does not represent any standard method of measuring frequency — the means of translating this figure in hertz is described in the machine code commentary.

Line 46010: The lower four bits of this register control the volume for all three voices. Fifteen is the maximum volume available.

Line 46020: The lower three bits of this register determine whether any of the voices are filtered. This POKE switches all the filters out.

Lines 46030–46040: These lines set up a note 'shape' which rises immediately from nothing, continues as a flat tone and dies out instantaneously, in other words a flat 'tone'.

Lines 46050–46070: The frequency of a note to be played is a value in the range 0–65535 and must therefore be held in two-byte form. The frequency control registers are two registers for each voice. These lines place the frequency specified by FR into the registers for voice one.

Line 46080: The waveform to be produced is controlled by the fourth register for the voice, with the value 16 setting bit four and dictating a triangle waveform. Setting bit zero (by POKEing 17 rather than 16), switches on the voice to produce the note specified.

Line 46100: A timing loop. The SID will go on producing the specified note until the loop is ended after NL repetitions.

Line 46200: POKEing zero into the same register as was used in line 46080 switches off bit zero and turns off the voice, ending the tone.

Beep — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $CC00
CC00               30 SYM
CC00               80 SID     = $D400
CC00               90 GETWRD  = $C000
CC00              100 COMMA   = $AEFD
CC00              110 DELAY   = $EEB3
CC00   A900       160 BEEP  LDA #0
CC02   8D04D4     170 STA SID+$04
CC05   8D0BD4     180 STA SID+$0B
CC08   8D12D4     190 STA SID+$12
CC0B   A90F       210 LDA #$0F
CC0D   8D18D4     220 STA SID+$18
CC10   A908       240 LDA #$08
CC12   8D17D4     250 STA SID+$17
CC15   A900       270 LDA #$00
CC17   8D05D4     280 STA SID+$05
CC1A   A9F0       290 LDA #$F0
CC1C   8D06D4     300 STA SID+$06
CC1F   2000C0     320 JSR GETWRD
CC22   A514       330 LDA $14
CC24   48         340 PHA
CC25   A515       350 LDA $15
CC27   48         360 PHA
CC28   20FDAE     370 JSR COMMA
CC2B   2000C0     380 JSR GETWRD
CC2E   68         390 PLA
CC2F   8560       400 STA $60
CC31   68         410 PLA
CC32   855F       420 STA $5F
CC34   0560       440 ORA $60
```

```
CC36  F021      450 BEQ L001
CC38  A514      460 LDA $14
CC3A  A615      470 LDX $15
CC3C  8D00D4    480 STA SID+$00
CC3F  8E01D4    490 STX SID+$01
CC42  A911      510 LDA #$11
CC44  8D04D4    520 STA SID+$04
CC47  20B3EE    540 L000 JSR DELAY
CC4A  A55F      560 LDA $5F
CC4C  C65F      570 DEC $5F
CC4E  AA        580 TAX
CC4F  D0F6      590 BNE L000
CC51  C660      600 DEC $60
CC53  A560      610 LDA $60
CC55  C9FF      620 CMP #$FF
CC57  D0EE      630 BNE L000
CC59  A900      650 L001 LDA #$00
CC5B  8D04D4    660 STA SID+$04
CC5E  60        670 RTS
CC5F            680 END
 TOTAL ERRORS IN FILE --- 0

SID             D400
GETWRD          C000
COMMA           AEFD
DELAY           EEB3
BEEP            CC00
L000            CC47
L001            CC59
 TOTAL NUMBER OF SYMBOLS --- 7
```

Machine code

```
ADD   DATA                          CHECKSUM

CC00  A9 00 8D 04 D4 8D 0B D4   701E
CC08  8D 12 D4 A9 0F 8D 18 D4   73C0
CC10  A9 08 8D 17 D4 A9 00 8D   7361
CC18  05 D4 A9 F0 8D 06 D4 20   61E8
CC20  00 C0 A5 14 48 A5 15 48   4B26
CC28  20 FD AE 20 00 C0 68 85   6B55
CC30  60 68 85 5F 05 60 F0 21   6439
CC38  A5 14 A6 15 8D 00 D4 8E   742E
CC40  01 D4 A9 11 8D 04 D4 20   51F0
CC48  B3 EE A5 5F C6 5F AA D0   B960
CC50  F6 C6 60 A5 60 C9 FF D0   CBC2
CC58  EE A9 00 8D 04 D4 60      5720
```

Commentary

Lines 160–190: Zero is loaded into the accumulator and then stored in the control registers for the three voices.

Lines 210–220: BEEP is always executed with volume at maximum, so the value 15 is stored in the SID volume control register.

Lines 240–250: Switch off filters on voices one, two and three.

Lines 260–300: These lines set up the envelope for the note to be BEEPed. Since all that is required is a tone of specified duration, this is achieved by setting the attack, decay and release to minimum. These three settings ensure a note which begins and ends without tapering up or down and is consistent throughout its duration.

Lines 320–420: The length of the note and its pitch value are picked up using GETWRD and stored.

Lines 440–450: If the note length at this point is zero, then an exit is made from the routine.

Lines 460–490: The frequency value is transferred to the frequency control register for voice one.

Lines 510–520: 17 is stored into the SID register at $D404, switching on voice one with a sawtooth waveform.

Lines 540–630: These lines begin by calling up the ROM routine DELAY, which produces a pause of approximately one millisecond. On returning from DELAY, the value in $5F–60 which represents the specified length of the note is decremented by one and the loop from line 540 repeated if zero has not been reached.

Lines 650–660: Zero is loaded into the SID register at $D404, thus switching off voice one.

Testing

BASIC: The routine is self-testing and should produce a single short tone when RUN.

MACHINE CODE: Turn up the sound on your TV to roughly half volume, and then enter the following line of BASIC in direct mode:

SYS (52224) 500, 10000

The result should be a note with a duration of some half a second.

Syntax

The syntax for BEEP is:

SYS (52224) { NOTE LENGTH} ,{ PITCH}

NOTE LENGTH may be any value in the range 0–65535, representing a maximum note length of 65 seconds approximately. Note that, while a note is being sounded, the execution of the BASIC program stops. It would be quite possible to set the beep for an indefinite period using the first half of the routine and to switch it off with a separate call to the second half. This would allow other processing to be done during the sounding of the note and may be advantageous in some circumstances.

The PITCH value may be in the range 0–65535 and is not the frequency of the note but, rather, represents a value of 0.06097 times the frequency in hertz. For example, if the pitch parameter were 1000, the frequency resulting would be 1000*0.06097, or approximately 61 hertz.

The command can be used simply to produce a pause in the execution of BASIC if the note value specified is zero.

3. CHORD

Having used one of the voices in the relatively straightforward BEEP command, it is only a small step to combine three simultaneous BEEPs into a CHORD. The method is identical to that used for BEEP, except that extra parameters have to be picked up for the number of voices to be employed and the pitch for each voice.

Chord — BASIC listing

```
47000 REM*******************************
47001 REM CHORD
47002 REM*******************************
47010 POKE SID+24,15
47020 POKE SID+23,8
47025 FOR I=0 TO 7*(NV-1) STEP 7
47030 POKE SID+5+I,0
47040 POKE SID+6+I,240
47050 HF(I/7)=INT(FR(I/7)/256)
47060 POKE SID+I,FR(I/7)-256*HF(I/7)
47070 POKE SID+1+I,HF(I/7)
47075 NEXT I
47077 FOR I=0 TO 7*(NV-1) STEP 7
47080 POKE SID+4+I,17
47090 NEXT I
47100 FOR I=1 TO NL : NEXT I
47190 FOR I=0 TO 7*(NV-1) STEP 7
47200 POKE SID+4+I,0
47210 NEXT I
47999 RETURN
```

Commentary

Line 47025: This loop ensures that the two POKEs which set up the shape of

the flat tone and its frequency will be carried out on the registers of as many voices (1–3) as are specified in the control module by the variable NV.

Lines 47050–47070: Equivalent to BASIC lines 46050–47070 in the BEEP command, except that an array is used so that the frequency values for all three voices can be specified.

Line 47077: The triangle waveform is switched on in each of the three voices.

Lines 47190–47210: The three voices are switched off.

The following changes must also be made to the control module. Add the new lines:

45140 FR(0) = 8000
45150 FR(1) = 10000
45160 FR(2) = 12000
45170 NV = 3

and change line 45200 to read:

45200 GOSUB 47000

Chord — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $CC5F
CC5F               30 SYM
CC5F               80 SID     = $D400
CC5F               90 GETWRD  = $C000
CC5F              100 COMMA   = $AEFD
CC5F              110 DOBEEP  = $CC47
CC5F              120 GETBYT  = $C006
CC5F   A900       150 CHORD LDA #00
CC61   8D04D4     160 STA SID+$04
CC64   8D0BD4     170 STA SID+$0B
CC67   8D12D4     180 STA SID+$12
CC6A   A90F       200 LDA #$0F
CC6C   8D18D4     210 STA SID+$18
CC6F   A908       230 LDA #$08
CC71   8D17D4     240 STA SID+$17
CC74   A900       260 LDA #00
CC76   8D05D4     270 STA SID+$05
CC79   8D0CD4     280 STA SID+$0C
CC7C   8D13D4     290 STA SID+$13
CC7F   A9F0       300 LDA #$F0
CC81   8D06D4     310 STA SID+$06
CC84   8D0DD4     320 STA SID+$0D
CC87   8D14D4     330 STA SID+$14
CC8A   2006C0     350 JSR GETBYT
CC8D   C901       360 CMP #$01
```

```
CC8F  905B      370 BCC IQERR
CC91  C904      380 CMP #$04
CC93  B057      390 BCS IQERR
CC95  85FE      400 STA $FE
CC97  48        410 PHA
CC98  20FDAE    420 JSR COMMA
CC9B  2000C0    440 JSR GETWRD
CC9E  A514      450 LDA $14
CCA0  48        460 PHA
CCA1  A515      470 LDA $15
CCA3  48        480 PHA
CCA4  A00E      500 LDY #$0E
CCA6  84FD      510 L000 STY $FD
CCA8  20FDAE    520 JSR COMMA
CCAB  2000C0    530 JSR GETWRD
CCAE  A4FD      540 LDY $FD
CCB0  A514      550 LDA $14
CCB2  9900D4    560 STA SID.Y
CCB5  A515      570 LDA $15
CCB7  9901D4    580 STA SID+1.Y
CCBA  98        590 TYA
CCBB  38        600 SEC
CCBC  E907      610 SBC #$07
CCBE  A8        620 TAY
CCBF  C6FE      630 DEC $FE
CCC1  D0E3      640 BNE L000
CCC3  68        660 PLA
CCC4  8560      670 STA $60
CCC6  68        680 PLA
CCC7  855F      690 STA $5F
CCC9  68        700 PLA
CCCA  AA        710 TAX
CCCB  A55F      730 LDA $5F
CCCD  0560      740 ORA $60
CCCF  F01A      750 BEQ L002
CCD1  A00E      760 LDY #$0E
CCD3  A911      770 L001 LDA #$11
CCD5  9904D4    780 STA SID+$4.Y
CCD8  98        790 TYA
CCD9  38        800 SEC
CCDA  E907      810 SBC #$07
CCDC  A8        820 TAY
CCDD  CA        830 DEX
CCDE  D0F3      840 BNE L001
CCE0  2047CC    850 JSR DOBEEP
CCE3  A900      860 LDA #$00
CCE5  8D0BD4    870 STA SID+$B
CCE8  8D12D4    880 STA SID+$12
CCEB  60        890 L002 RTS
CCEC  4C48B2    900 IQERR JMP $B248
CCEF            910 END
 TOTAL ERRORS IN FILE --- 0
```

```
SID               D400
GETWRD            C000
COMMA             AEFD
DOBEEP            CC47
GETBYT            C006
CHORD             CC5F
L000              CCA6
L001              CCD3
L002              CCEB
IQERR             CCEC
 TOTAL NUMBER OF SYMBOLS --- 10
```

Machine code

```
ADD   DATA                          CHECKSUM

CC5F  A9 00 8D 04 D4 8D 0B D4   701E
CC67  8D 12 D4 A9 0F 8D 18 D4   73C0
CC6F  A9 08 8D 17 D4 A9 00 8D   7361
CC77  05 D4 8D 0C D4 8D 13 D4   53AE
CC7F  A9 F0 8D 06 D4 8D 0D D4   AC42
CC87  8D 14 D4 20 06 C0 C9 01   6CC3
CC8F  90 5B C9 04 B0 57 85 FE   8104
CC97  48 20 FD AE 20 00 C0 A5   59A5
CC9F  14 48 A5 15 48 A0 0E 84   3750
CCA7  FD 20 FD AE 20 00 C0 A4   B424
CCAF  FD A5 14 99 00 D4 A5 15   B87F
CCB7  99 01 D4 98 38 E9 07 A8   76DA
CCBF  C6 FE D0 E3 68 85 60 68   D12C
CCC7  85 5F 68 AA A5 5F 05 60   78EE
CCCF  F0 1A A0 0E A9 11 99 04   9A22
CCD7  D4 98 38 E9 07 A8 CA D0   AACC
CCDF  F3 20 47 CC A9 00 8D 0B   9D8D
CCE7  D4 8D 12 D4 60 4C 48 B2   A232
CCEF                            0000
```

Commentary

Lines 150–180: All voices are set to off.

Lines 200–210: Volume is set to maximum.

Lines 230–240: Switch off filters.

Lines 260–330: The envelope for all three voices is set up as it was for voice one only in BEEP.

Lines 350–410: An addition to the BEEP command, the number of voices specified is checked to see that it falls in the range 1–3. If the number is valid it is stored temporarily on the stack.

Lines 440–480: The length of the note(s) is obtained and stored on the stack.

Lines 500–640: The frequencies for the specified number of voices are picked up from the BASIC line and then stored in the appropriate SID registers. The number of voices has been left in $FE by the GETBYT routine and this is used as a counter to guide the command in knowing how many values are needed. Each time the loop is executed, seven is subtracted from the contents of the Y register so that, when this is used for indexed addressing, it will refer to the control registers for each of the three voices in turn.

Lines 660–710: The number of voices and the length of the notes are recalled from the stack. The number of voices is stored in the X register and the note length in $5F–60.

Lines 730–750: A test is made to ensure that the note length is not zero. If it is, then the routine is aborted.

Lines 760–840: The same process as for the frequencies, but this time the loop is turning on the sawtooth waveform in each of the specified number of voices, starting with voice three and subtracting seven for each iteration of the loop. These lines therefore begin the execution of the chord.

Line 850: This line calls up the timing section of BEEP.

Lines 860–920: These lines turn off voices two and three — voice one was turned off by the call to BEEP.

Line 900: A call to the ILLEGAL QUANTITY error message.

Testing

BASIC: The routine is self-testing. When RUN it should produce a distinct chord, ie three notes played simultaneously.

MACHINE CODE: The routine, as you will have seen from the commentary, is very much like BEEP, so entering:

```
SYS(52319) 1,500,10000
```

will produce exactly the same tone and duration as was heard in testing the earlier command. If no problem is encountered, you can go on to try:

```
SYS(52319) 3,500,1000,10000,60000
```

and now the original tone will be accompanied by a high pitched squeak and a low hum.

Syntax

The correct syntax for CHORD is:

SYS(52319){ VOICES} ,{ DURATION} ,{ PITCH1} ,[{ PITCH2} , { PITCH 3}]

VOICES may be any value in the range 1–3. DURATION is expressed in milliseconds and may be in the range 1–65535.

Note that a syntax error will be generated if the number of PITCH parameters specified in the command is not equal to the number of voices specified. In other respects, ie the value of pitch and duration, the command is identical in effect to BEEP.

4. ADVANCED SOUND COMMANDS

The two commands we have entered so far, useful though they may be for producing a few quick tones while a program is running, do not make anything like full use of the capabilities of the SID chip, which, as noted earlier, is a highly sophisticated sound synthesiser in itself. The collection of commands you are about to enter are another matter. With these commands, you will be able to enter sequences of notes for each of the three voices separately in the form of strings, with each note being given its straightforward name, ie a, b, c, d, e, f, g. The full eight octaves of the SID chip may be employed and all legal combinations of attack, sustain, decay, release and filter are available. As with all use of the SID chip, the procedure for this group of commands will be first to provide commands to set up the various sound parameters, and then to design a command to activate the notes specified.

In all of this, there will be one difficulty which dogs any effective use of the SID chip, and that is that the registers which it uses are write only — memory locations into which values may be POKEd without problem but which return the spurious value of 255 when read using PEEK. What this means is that, while it is perfectly possible to alter the whole value of a particular register using POKE, it is not possible to alter the condition of specified bits without affecting the others, since to do this requires that a value be obtained from the register with PEEK, manipulated and then returned to the register.

The problem is overcome in the commands that follow by the use of 'pseudo-registers', that is to say a series of locations in normal memory which will act as a parallel storage area for values which are to be placed in the real memory locations. Using the values stored in the pseudo-registers, we shall be able to perform any manipulations thought to be necessary and

then to place the resultant numbers into the SID registers. Note that the operations which we shall be performing most of the time will therefore have no effect whatsoever on the SID chip. POKEing the sound output bit on the correct pseudo-register will *not* switch on the sound output of the SID chip until the contents of the pseudo-register are copied into the genuine SID register.

This use of pseudo-registers not only overcomes the problem of the SID's write only registers, it also gives a degree of flexibility in programming since there is no need for us to observe any strict order in which values may be changed. Everything may be done for the convenience of the programmer since the SID chip will not be aware of any changes until the pseudo-registers are copied. In relation to the commands you have already entered, BEEP and CHORD, the use of pseudo-registers also has the advantage that these simple tone commands, while they change the SID registers, have no effect on the pseudo-registers. BEEP and CHORD may therefore be used in between separate uses of the more advanced sound commands without any need to set up all the parameters a second time in the pseudo-registers.

The location chosen for the pseudo-registers, 24 of them in all to correspond to the total number of registers for the SID chip, is $02E0–02F8 (736–760). The first pseudo-register, at $02E0 (736), thus corresponds to the first SID register, at $D400 (53248), while the final pseudo-register, at $02F8 (760), corresponds to the final SID register, at $D418 (53272). In examining the routines which follow, you will be able to use the description of the SID registers on page 461 of the Programmer's Reference Guide if you are uncertain of the function of any particular pseudo-register.

5. SOUND PARAMETERS (GETPAR)

The registers we shall be dealing with in the commands which follow almost all contain two four-bit values (or 'nibbles'), that is two values in the range 0–15, one in the upper four bits of the register (the high nibble) and one in the lower four (the low nibble). For that reason, it is possible to use a single routine to service all the commands by picking up sound parameters from the BASIC text.

The procedure is as follows:

1) Obtain the parameter which specifies the value of the nibble.
2) Obtain the voice number.
3) Calculate, according to the voice number, the relative position, or 'offset' of the registers for the particular voice, as compared to the start of the pseudo-registers. This will be either 0, 7 or 14.

GetPar — assembly language listing

```
ADD.  DATA     SOURCE CODE
```

```
00                    10 PRT
00                    20 ORG $CCEF
CCEF                  30 SYM
CCEF                  70 GETBYT = $C006
CCEF   2006C0        100 GETNYB JSR GETBYT
CCF2   C910          110 CMP #$10
CCF4   B001          120 BCS IQERR
CCF6   60            130 RTS
CCF7   4C48B2        140 IQERR JMP $B248
CCFA   2006C0        160 GETVOC JSR GETBYT
CCFD   A8            170 TAY
CCFE   C001          180 CPY #1
CD00   90F5          190 BCC IQERR
CD02   C004          200 CPY #4
CD04   B0F1          210 BCS IQERR
CD06   A900          220 LDA #0
CD08   88            230 L000 DEY
CD09   F005          240 BEQ L001
CD0B   18            250 CLC
CD0C   6907          260 ADC #7
CD0E   D0F8          270 BNE L000
CD10   A8            280 L001 TAY
CD11   60            290 RTS
CD12                 300 END
 TOTAL ERRORS IN FILE --- 0

GETBYT               C006
GETNYB               CCEF
IQERR                CCF7
GETVOC               CCFA
L000                 CD08
L001                 CD10
 TOTAL NUMBER OF SYMBOLS --- 6
```

Machine code

```
ADD    DATA                              CHECKSUM

CCEF   20 06 C0 C9 10 B0 01 60            39B2
CCF7   4C 48 B2 20 06 C0 A8 C0            5580
CCFF   01 90 F5 C0 04 B0 F1 A9            548B
CD07   00 88 F0 05 18 69 07 D0            4392
CD0F   F8 A8 60                           0590
```

Commentary

Lines 100–130: A check that the parameter falls within the range 0–15, ie that it is in the range of a four-bit number, or nibble.

Lines 160–210: The voice number is picked up and tested against the range of 1–3.

Lines 220–290: The voice number is converted into an offset, or start point for a set of registers in relation to the start of the SID registers. Voice one would produce an offset of zero, voice two of seven and voice three of fourteen.

Testing

This routine can only be effectively tested when one or more of the commands to follow have been entered.

6. VOLUME

Volume is an overall setting which affects the output of all three voice channels and can be set to anything in the range 0–15. A setting of fifteen sets all voices to maximum volume, while zero allows the voices to play as normal but nothing will be heard through the television speaker.

The procedure for VOLUME is as follows:

1) Obtain the volume parameter.
2) Place it into the pseudo-register corresponding to $D018 as the low nibble.

Volume — assembly language listing

```
ADD.  DATA      SOURCE CODE
00              10 PRT
00              20 ORG $CD12
CD12            30 SYM
CD12            70 BASE    = $02E0
CD12            80 GETNYB = $CCEF
CD12  ADF802    110 VOLUME LDA BASE+$18
CD15  2970      120 AND #$70
CD17  8DF802    130 STA BASE+$18
CD1A  20EFCC    140 JSR GETNYB
CD1D  0DF802    150 ORA BASE+$18
CD20  8DF802    160 STA BASE+$18
CD23  60        170 RTS
CD24            180 END
 TOTAL ERRORS IN FILE --- 0

BASE            2E0
GETNYB          CCEF
VOLUME          CD12
 TOTAL NUMBER OF SYMBOLS --- 3
```

Machine code

```
ADD   DATA                        CHECKSUM

CD12  AD F8 02 29 70 8D F8 02     9EF6
CD1A  20 EF CC 0D F8 02 8D F8     6FEA
CD22  02 60                       0064
```

Commentary

Lines 110–130: The first use of one of the pseudo-registers mentioned above in the introduction to the advanced sound commands. These lines load the contents of the nineteenth pseudo-register into the accumulator (register zero is the first) and sets the nibble relating to volume to zero. In addition, voice three is switched on.

Line 140: GETNYB is used to pick up the volume parameter from the BASIC program.

Lines 150–160: The volume setting is now placed into the pseudo-register without disturbing the upper three bits, which control aspects of the filtering of each voice. Note that the SID chip is entirely unaffected by this procedure since we are only working with the pseudo-registers at this point.

Testing

To test the routine without going on to enter the full range of sound commands, the following three lines of BASIC may be used:

```
10 POKE 760,0
20 SYS (52498) 15
30 PRINT PEEK (760) AND 15
```

The result printed on the screen should be 15. If it is not, remember that the error may lie *either* in the current command *or* in the GETNYB routine previously entered.

Syntax

The syntax for VOLUME is:

SYS (52498) { VOLUME SETTING}

where VOLUME SETTING may be in the range 0–15, with 15 being maximum volume and zero effectively cutting off the output of the three voices.

7. ADSR

If you have any interest in the use of the SID chip you will probably already know that the four characteristics of a single note — attack, decay, sustain and release — when taken as a group are known by their intitial letters, ADSR. Together they define almost everything that is important about a note apart from its pitch, the colour imparted to it by the use of filters and

its volume. The following command allows all four of these important characteristics to be set in one statement.

The procedure for ADSR is as follows:

1) Obtain the voice number.
2) Obtain the attack value and translate this into the form of a high nibble.
3) Obtain the decay value.
4) Place the combined value into the appropriate pseudo-register.
5) Repeat steps two, three and four for sustain and release.

ADSR — assembly language listing

```
ADD.  DATA      SOURCE CODE
00                10 PRT
00                20 ORG $CD24
CD24              30 SYM
CD24              80 COMMA  = $AEFD
CD24              90 GETNYB = $CCEF
CD24             100 GETVOC = $CCFA
CD24             110 BASE   = $02E0
CD24  20FDAE     140 GETTOP JSR COMMA
CD27  20EFCC     150 JSR GETNYB
CD2A  0A         160 ASL A
CD2B  0A         170 ASL A
CD2C  0A         180 ASL A
CD2D  0A         190 ASL A
CD2E  A4FE       200 LDY $FE
CD30  99E502     210 STA BASE+5.Y
CD33  60         220 RTS
CD34  20FACC     250 ASDR  JSR GETVOC
CD37  84FE       260 STY $FE
CD39  2024CD     270 JSR GETTOP
CD3C  E6FE       280 INC $FE
CD3E  2024CD     290 JSR GETTOP
CD41  C6FE       300 DEC $FE
CD43  2048CD     310 JSR GETLOW
CD46  E6FE       320 INC $FE
CD48  20FDAE     340 GETLOW JSR COMMA
CD4B  20EFCC     350 JSR GETNYB
CD4E  A4FE       360 LDY $FE
CD50  19E502     370 ORA BASE+5.Y
CD53  99E502     380 STA BASE+5.Y
CD56  60         390 RTS
CD57             400 END
  TOTAL ERRORS IN FILE --- 0

COMMA             AEFD
GETNYB            CCEF
```

```
GETVOC              CCFA
BASE                2E0
GETTOP              CD24
ASDR                CD34
GETLOW              CD48
 TOTAL NUMBER OF SYMBOLS --- 7
```

Machine code

```
ADD   DATA                              CHECKSUM

CD24  20 FD AE 20 EF CC 0A 0A    71C6
CD2C  0A 0A A4 FE 99 E5 02 60    34A0
CD34  20 FA CC 84 FE 20 24 CD    79C5
CD3C  E6 FE 20 24 CD C6 FE 20    C45C
CD44  48 CD E6 FE 20 FD AE 20    8A50
CD4C  EF CC A4 FE 19 E5 02 99    D3D9
CD54  E5 02 60                   03F8
```

Commentary

Lines 140–190: A subroutine — the main program begins at line 250.

Lines 150–190: The parameter for either attack or sustain is picked up from the BASIC program, according to the point from which the subroutine is called in the main routine. The reason that the lines refer only to attack and sustain is that these two are stored in the high nibbles of the two ADSR registers. The effect of the lines is to take the 0–15 value and to shift it left so that it will fit into the upper rather than the lower four bytes of the register.

Lines 200–210: Storage is done using the Y register to perform indexed addressing, based on the offset stored at $FE. This offset, which works out on the basis of the voice number which of the three groups of seven registers is to be affected, was calculated by the part of the sound parameters routine and the result was stored at $FE by the main routine which follows.

Lines 250–260: The call to part of the sound parameters routine which obtains the voice number and calculates the start position of the registers referring to it. The result is stored at $FE temporarily.

Line 270: The earlier subroutine is called to pick up the ATTACK value and place it in the high nibble of the appropriate pseudo-register.

Lines 280–290: The value of the offset is increased by one so that it points to the next register. Another call to GETTOP now obtains the SUSTAIN

setting and places it into the high nibble of the register following that for ATTACK.

Lines 300–310: The offset is decremented so that it once again indicates the first of the two registers being used and then the GETLOW subroutine is called. This first call to the subroutine, which follows immediately after these lines, obtains the RELEASE value from BASIC and places it straight into the low nibble of the first register.

Lines 340–390: The GETLOW subroutine which places a value in the range 0–15 in the low nibble of the appropriate register.

Testing

In testing this routine, you might like to remember that this is the first use we have made of the GETVOC routine, so, if a problem is encountered, it would be wise to check both sections of code. A simple BASIC test for the current routine is:

```
10 POKE 748,0 : POKE 749,0
20 SYS (52532) 2,15,15,15,15,
30 PRINT PEEK(748),PEEK(749)
```

The result of running this little program should be the display of two 255s on the screen, indicating that the high and low nibbles of both the relevant registers for voice two have been set to 15, ie every bit is set in both registers.

Syntax

The syntax for ADSR is:

SYS (52532) { VOICE} ,{ ATTACK} ,{ DECAY} ,{ SUSTAIN} , { RELEASE}

where VOICE must be in the range 1–3 and all the other parameters must be in the range 0–15.

8. FILTERS

The command you are about to enter allows you to set up three important parameters of the SID's filters, namely the type of filter, the filter resonance factor and the filter breakthrough. The filters, once set, cannot be varied from voice to voice, though for each voice there is a choice as to whether the current filter setting will be applied or not. This 'on/off' capability will be dealt with under the next command — at the present time, we shall only be concerned with the setting up of the filters, not with applying them to the three voices.

The filters which may be applied fall into four different categories, bandpass, highpass, lowpass and notch. Before any of these filter types will work, there must also be a 'break frequency' specified and provision is made for this in the syntax for the command. The break frequency simply represents a significant frequency value and it will be used in a different way by each of the filters. Given a break frequency, the effects of the filter types are as follows:

a) **Bandpass:** Frequencies at or around the break frequency are allowed through at full strength but frequencies further from the break frequency (either up or down) are attenuated.
b) **Highpass:** Frequencies above the break frequency are allowed through at full strength but those below are attenuated.
c) **Lowpass:** Frequencies below the break frequency are allowed past at full strength but those above are attenuated.
d) **Notch:** All frequencies are allowed through at full strength with the exception of a narrow band around the break frequency.

In addition to the setting of the filter type and the break frequency, the current command also allows the setting of the filter resonance, with the note becoming more emphatic as the resonance is increased.

The procedure for FILTER is as follows:

1) Obtain the filter type.
2) Obtain the resonance parameter.
3) Obtain the filter breakpoint frequency parameter.
4) Place the frequency value into the pseudo-registers corresponding to $D015–6.
5) Calculate the correct 'bit pattern' to set up the specified filter type and place this in the pseudo-register corresponding to $D018.

Filter — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $CD57
CD57               30 SYM
CD57               90 GETBYT = $C006
CD57              100 COMMA  = $AEFD
CD57              110 GETNYB = $CCEF
CD57              120 BASE   = $02E0
CD57              130 GETWRD = $C000
CD57              140 IQERR  = $CCF7
CD57   2006C0     170 FILTER JSR GETBYT
CD5A   C904       180 CMP #$04
CD5C   B099       190 BCS IQERR
CD5E   48         200 PHA
CD5F   20FDAE     210 JSR COMMA
```

```
CD62  20EFCC     220 JSR GETNYB
CD65  48         230 PHA
CD66  20FDAE     240 JSR COMMA
CD69  2000C0     250 JSR GETWRD
CD6C  A514       260 LDA $14
CD6E  A615       270 LDX $15
CD70  8DF502     280 STA BASE+$15
CD73  8EF602     290 STX BASE+$16
CD76  ADF702     300 LDA BASE+$17
CD79  290F       310 AND #$F
CD7B  8DF702     320 STA BASE+$17
CD7E  68         330 PLA
CD7F  0A         340 ASL A
CD80  0A         350 ASL A
CD81  0A         360 ASL A
CD82  0A         370 ASL A
CD83  0DF702     380 ORA BASE+$17
CD86  8DF702     390 STA BASE+$17
CD89  68         400 PLA
CD8A  AA         410 TAX
CD8B  ADF802     420 LDA BASE+$18
CD8E  290F       430 AND #$F
CD90  1D97CD     440 ORA TABLE.X
CD93  8DF802     450 STA BASE+$18
CD96  60         460 RTS
CD97  204010     470 TABLE BYT $20.$40.$1
0.$50
CD9B             480 END
 TOTAL ERRORS IN FILE --- 0

GETBYT           C006
COMMA            AEFD
GETNYB           CCEF
BASE             2E0
GETWRD           C000
IQERR            CCF7
FILTER           CD57
TABLE            CD97
 TOTAL NUMBER OF SYMBOLS --- 8
```

Machine code

```
ADD   DATA                        CHECKSUM

CD57  20 06 C0 C9 04 B0 99 48     3A6A
CD5F  20 FD AE 20 EF CC 48 20     7258
CD67  FD AE 20 00 C0 A5 14 A6     B762
CD6F  15 8D F5 02 8E F6 02 AD     5579
CD77  F7 02 29 0F 8D F7 02 68     8AC0
CD7F  0A 0A 0A 0A 0D F7 02 8D     0E35
CD87  F7 02 68 AA AD F8 02 29     9D15
CD8F  0F 1D 97 CD 8D F8 02 60     371C
CD97  20 40 10 50                 0270
```

Commentary

Lines 170–200: Check that the filter type is in the range 1–3 and then save it temporarily on the stack.

Lines 220–230: Pick up the resonance parameter and save it on the stack.

Lines 250–290: Pick up the break frequency parameter using GETWRD and store this two-byte value in the appropriate pseudo-registers.

Lines 300–320: The current resonance setting (in the pseudo-registers) is cleared by ANDing the register with 15 — the resonance value is stored in the high nibble.

Lines 330–390: The new resonance value is recalled from the stack, shifted left so that it now occupies the upper four bits and then stored in the appropriate register.

Lines 400–410: The filter type parameter is recalled from the stack and stored in the X register.

Lines 420–430: The number of the present filter type is cleared by the use of AND without affecting the high nibble of the register.

Lines 440–450: A look-up table is now used to transform the filter type which was specified in the BASIC program into the correct low nibble to produce the desired filtering. The result is stored back into the appropriate register.

Line 470: The table representing the actual values which are needed to produce each of the filter types.

Testing

The following three lines of BASIC will suffice to make a preliminary test of the command:

```
10 FOR I = 0 TO 3 : POKE 757 + I,0 : NEXT
20 SYS (52567) 1,10,513
30 FOR I = 0 TO 3 : PRINT PEEK (757 + I) : NEXT
```

The result of running this program should be that the following four values are printed on the screen:

```
1
2
160
64
```

Syntax

The syntax for FILTER is:

SYS (52567) { TYPE} ,{ RESONANCE} ,{ BREAK}

TYPE is a number in the range 0–3, as follows:

0 = BANDPASS
1 = HIGHPASS
2 = LOWPASS
3 = NOTCH

RESONANCE is a number in the range 0–15. BREAK is a number in the range 0–65535, though only values in the range 0–16384 will be significant to the system. In translating this break parameter into a frequency in hertz, break must be multiplied by 0.06097, as in the BEEP and CHORD commands.

9. VOICE CONTROL (VOICE)

There remain a number of major functions in relation to the three voices which we have not yet dealt with. All of them are functions controlled by a single bit and, rather than include a separate command for each, they have all been gathered together into a single command which is a trifle unwieldy at first sight but which will cause no problems in practice.

The majority of the parameters we are concerned with in this command are controlled by a single register for each voice, the fifth of the registers in the group allocated to each voice. The functions of the individual bits in this register are:

BIT	FUNCTION
0	Start attack cycle when one, start decay cycle when zero.
1	Synchronise to other voices.
2	Ring modulate with another voice.
3	Should always be set to zero.
4	Triangle waveform when set.
5	Sawtooth waveform when set.
6	Pulse waveform when set.
7	Noise waveform when set.

In addition to these functions, there is also the question of the decision as to whether each voice will have the current filters applied to it or not. This is controlled by the lower three bits of the eighteenth register, with bit zero dictating that voice one will be filtered if it is set, that it will be unfiltered if reset, and so on. The current command therefore requires three pieces of information:

1) The voice number.

2) Whether it is to be filtered, in the form of a one or a zero.
3) The value of the control register. This will be a value between zero and 255 which you must calculate according to which of the bits you wish to have set. Alternatively, you can use an expression such as (2^2+2^4), where a power of two is used to refer to each of the bits to be set. In the case of the example expression given here, bits two and four would be set.

Voice — assembly language listing

```
ADD.  DATA      SOURCE CODE
00              10 PRT
00              20 ORG $CD9B
CD9B            30 SYM
CD9B            80 GETVOC = $CCFA
CD9B            90 COMMA  = $AEFD
CD9B           100 GETBYT = $C006
CD9B           110 GETWRD = $C000
CD9B           120 BASE   = $02E0
CD9B  20FACC   150 VOICE JSR GETVOC
CD9E  84FE     160 STY $FE
CDA0  A904     180 LDA #4
CDA2  C007     190 CPY #7
CDA4  F005     200 BEQ L000
CDA6  C00E     210 CPY #$E
CDA8  F002     220 BEQ L001
CDAA  4A       230 LSR A
CDAB  4A       240 L000 LSR A
CDAC  85FD     250 L001 STA $FD
CDAE  49FF     260 EOR #$FF
CDB0  2DF702   280 AND BASE+$17
CDB3  8DF702   290 STA BASE+$17
CDB6  20FDAE   300 JSR COMMA
CDB9  2000C0   310 JSR GETWRD
CDBC  A514     330 LDA $14
CDBE  0515     340 ORA $15
CDC0  F008     350 BEQ L002
CDC2  A5FD     360 LDA $FD
CDC4  0DF702   380 ORA BASE+$17
CDC7  8DF702   390 STA BASE+$17
CDCA  20FDAE   400 L002 JSR COMMA
CDCD  2006C0   420 JSR GETBYT
CDD0  A4FE     430 LDY $FE
CDD2  99E402   440 STA BASE+$04.Y
CDD5  60       450 RTS
CDD6           460 END
 TOTAL ERRORS IN FILE --- 0

GETVOC          CCFA
COMMA           AEFD
GETBYT          C006
```

```
GETWRD              C000
BASE                2E0
VOICE               CD9B
L000                CDAB
L001                CDAC
L002                CDCA
 TOTAL NUMBER OF SYMBOLS --- 9
```

Machine code

```
ADD    DATA                          CHECKSUM

CD9B   20 FA CC 84 FE A9 04 C0        7B9C
CDA3   07 F0 05 C0 0E F0 02 4A        509E
CDAB   4A 85 FD 49 FF 2D F7 02        750C
CDB3   8D F7 02 20 FD AE 20 00        9160
CDBB   C0 A5 14 05 15 F0 08 A5        912D
CDC3   FD 0D F7 02 8D F7 02 20        A928
CDCB   FD AE 20 06 C0 A4 FE 99        B985
CDD3   E4 02 60                       03F4
```

Commentary

Lines 150–160: Pick up the voice number and store it in $FE.

Lines 180–250: On the basis of the voice number, these lines determine which bit is of interest in determining whether the voice is to be filtered. A mask is then constructed consisting of a byte in which only the relevant bit is set. This is done by using the offset value returned in the Y register by the GETVOC routine. The accumulator is loaded with four, ie a mask is created with only bit two set. The value in the Y register is then compared with seven and 14, the two possible offsets apart from zero. If the offset is seven, the voice is two and the mask is shifted to the right once thus moving the set bit to position one. If the offset is 14, the voice is three and the mask does not have to be rotated at all. If the offset is neither seven nor 14, the voice specified must have been one and the mask must be shifted left twice to move the set bit into position zero. The final mask is stored at $FD.

Lines 260–290: Another mask is now created consisting of a byte where all the bits are set *except* the one referring to the filtering of the current voice. The appropriate register is ANDed with this mask so that the bit referring to the current voice is turned off.

Lines 310–390: Having created the necessary mask, based on the voice number, and reset the relevant bit, we now need to know whether the voice is to be filtered or not. GETWRD is used to pick up the parameter which will specify this. If the parameter picked up is zero, then no action will be taken since the filter is not required and we have already cleared the relevant filter bit. If the parameter is greater than zero then the original mask,

in which the relevant bit is set, is used to switch on the correct filter bit in the pseudo-register.

Lines 400–450: All that remains is to pick up the control register value and store it in the appropriate voice control pseudo-register.

Testing

The following three lines of BASIC will provide an adequate preliminary test for the routine:

```
10 POKE 754,0 : POKE 759,0
20 SYS (52635) 3,1,170
30 PRINT PEEK(754), PEEK(759)
```

The result of running this should be the display of:

```
170     4
```

Syntax

The syntax for VOICE is:

SYS (52635) { VOICE} ,{ FILTER} ,{ REGISTER}

VOICE is a number in the range 1–3. FILTER is a number in the range 0–65535, where zero will result in the specified voice being unfiltered and other values in it being filtered. REGISTER is the value to be stored in the main control register for the voice. The relevance of the various bits and the methods of calculating the necessary control value are explained in the introduction above.

10. PULSE

Of the waveforms available through the SID one, the pulse waveform has an extra parameter attached, the width of the pulse (see the Programmer's Reference Guide for a more detailed explanation of the waveform itself). The current command, PULSE, simply allows this pulse width to be defined for later use by the PLAY command.

The procedure for PULSE is as follows:

1) Obtain the voice parameter.
2) Obtain the pulse width parameter.
3) Place the pulse width parameter into pseudo-registers 2–3 for the specified voice.

Pulse — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $CF64
CF64               30 SYM
CF64               40 GETVOC = $CCFA
```

```
CF64                50 COMMA  = $AEFD
CF64                60 GETWRD = $C000
CF64                65 BASE   = $02E0
CF64                70 PULSE
CF64  20FACC        80 JSR GETVOC
CF67  84FE          90 STY $FE
CF69  20FDAE       100 JSR COMMA
CF6C  2000C0       105 JSR GETWRD
CF6F  A4FE         110 LDY $FE
CF71  A514         120 LDA $14
CF73  99E202       130 STA BASE+2.Y
CF76  A515         140 LDA $15
CF78  99E302       150 STA BASE+3.Y
CF7B  60           160 RTS
CF7C               170 END
 TOTAL ERRORS IN FILE --- 0

GETVOC             CCFA
COMMA              AEFD
GETWRD             C000
BASE               2E0
PULSE              CF64
 TOTAL NUMBER OF SYMBOLS --- 5
```

Machine code

```
ADD   DATA                          CHECKSUM
CF64  20 FA CC 84 FE 20 FD AE        7B58
CF6C  20 00 C0 A4 FE A5 14 99        3D85
CF74  E2 02 A5 15 99 E3 02 60        9028
CF7C
```

Commentary

Lines 80–90: The voice parameter is picked up and the offset for the voice stored at $FE.

Line 105: The pulse width parameter is obtained by use of GETWRD.

Lines 120–150: The pulse width value is stored in the second and third pseudo-registers of the appropriate voice.

Testing

The routine cannot be tested properly until the PLAY command has been entered.

Syntax

The syntax for PULSE is:

SYS (53092) { VOICE} { PULSE WIDTH}

PULSE WIDTH may be any value in the range 0–65535 but only values in

the range 0–4095 will be significant to the SID.

11. PLAY

After all the sound commands that so far haven't done anything we come to the finale — the PLAY command which will allow you to see if all your careful setting of parameters has been worthwhile. The object of PLAY is to take the chore out of composing and playing music on the 64. Gone are the long and meaningless tables of DATA statements providing the values to be POKEd into the SID registers. In their place you can now use up to three strings whose letters represent straightforward musical notes in any of the eight octaves the SID can generate. In other words, provided that you can translate sheet music into the letters 'A' to 'G' you can get right down to programming your favourite music in three parts. The special effects such as synthesised pianos and glockenspiels you will have to leave to the other commands, but it is PLAY that will do the real work and open up the SID for you.

Such is the length of the command that, rather than attempting to give a brief summary of the procedure, we strongly suggest that you read the commentary in conjunction with the listing, thoroughly, *before* beginning to enter the routine.

Play — assembly language listing

```
ADD.   DATA      SOURCE CODE
00                 10 PRT
00                 20 ORG $CDD6
CDD6               30 SYM
CDD6              110 DELAY  = $EEB3
CDD6              120 BASE   = $02E0
CDD6              130 SID    = $D400
CDD6              140 GETBYT = $C006
CDD6              150 GETWRD = $C000
CDD6              160 GETSTR = $C089
CDD6              170 DOVOC  = $CCFD
CDD6              180 COMMA  = $AEFD
CDD6  B565        210 GETCHR LDA $65.X
CDD8  F00A        230 BEQ L000
CDDA  D665        250 DEC $65.X
CDDC  A166        260 LDA ($66.X)
CDDE  F666        280 INC $66.X
CDE0  D002        290 BNE L000
CDE2  F667        300 INC $67.X
CDE4  60          310 L000 RTS
CDE5  20D6CD      350 GETNOT JSR GETCHR
CDE8  F037        360 BEQ NONOTE
CDEA  C94F        380 CMP #79
CDEC  F036        390 BEQ OCTAVE
CDEE  A006        410 LDY #6
```

```
CDF0  D937CE     420 L001 CMP TABLE.Y
CDF3  F009       430 BEQ NOTE
CDF5  88         440 DEY
CDF6  10F8       450 BPL L001
CDF8  204ACF     470 IQERR JSR OFF
CDFB  4C48B2     480 JMP $B248
CDFE  C8         500 NOTE INY
CDFF  C006       510 CPY #6
CE01  B00E       520 BCS EXIT
CE03  20D6CD     530 JSR GETCHR
CE06  F009       540 BEQ EXIT
CE08  C923       560 CMP #35
CE0A  D007       580 BNE BACKUP
CE0C  98         590 TYA
CE0D  18         600 CLC
CE0E  6907       610 ADC #7
CE10  A8         620 TAY
CE11  98         640 EXIT TYA
CE12  60         650 RTS
CE13  B566       670 BACKUP LDA $66.X
CE15  D002       680 BNE L002
CE17  D667       690 DEC $67.X
CE19  D666       700 L002 DEC $66.X
CE1B  F665       710 INC $65.X
CE1D  4C11CE     720 JMP EXIT
CE20  98         740 TYA
CE21  A000       750 NONOTE LDY #00
CE23  60         770 RTS
CE24  20D6CD     790 OCTAVE JSR GETCHR
CE27  F0CF       800 BEQ IQERR
CE29  C930       820 CMP #48
CE2B  90CB       830 BCC IQERR
CE2D  C938       840 CMP #56
CE2F  B0C7       850 BCS IQERR
CE31  290F       870 AND #$0F
CE33  9523       880 STA $23.X
CE35  90AE       900 BCC GETNOT
CE37  434446     920 TABLE BYT 67.68.70.7
1.65.69.66
CE3E  88         950 CONVER DEY
CE3F  B958CE     960 LDA NTABLO.Y
CE42  8514       970 STA $14
CE44  B964CE     980 LDA NTABHI.Y
CE47  8515       990 STA $15
CE49  B523      1000 LDA $23.X
CE4B  A8        1010 TAY
CE4C  C007      1020 CLOOP CPY #7
CE4E  B007      1030 BCS L003
CE50  4615      1040 LSR $15
CE52  6614      1050 ROR $14
CE54  C8        1060 INY
```

```
CE55  D0F5      1070 BNE CLOOP
CE57  60        1080 L003 RTS
CE58  1E8B06    1100 NTABLO BYT 030.139.0
06.243.143.250.46
CE5F  187EAC    1110 BYT 024.126.172.230.
248
CE64  8696B3    1130 NTABHI BYT 134.150.1
79.200.225.168.253
CE6B  8E9FBD    1140 BYT 142.159.189.212.
238
CE70            1170 PLAY
CE70  204ACF    1190 JSR OFF
CE73  ADE402    1210 LDA BASE+4
CE76  0901      1220 ORA #1
CE78  8DE402    1230 STA BASE+4
CE7B  ADEB02    1240 LDA BASE+$B
CE7E  0901      1250 ORA #1
CE80  8DEB02    1260 STA BASE+$B
CE83  ADF202    1270 LDA BASE+$12
CE86  0901      1280 ORA #1
CE88  8DF202    1290 STA BASE+$12
CE8B  2006C0    1310 JSR GETBYT
CE8E  C900      1320 CMP #0
CE90  F066      1330 BEQ IQERR
CE92  C904      1340 CMP #4
CE94  B062      1350 BCS IQERR
CE96  85FE      1360 STA $FE
CE98  20FDAE    1370 JSR COMMA
CE9B  2000C0    1390 JSR GETWRD
CE9E  A514      1400 LDA $14
CEA0  48        1410 PHA
CEA1  A515      1420 LDA $15
CEA3  48        1430 PHA
CEA4  A5FE      1440 LDA $FE
CEA6  85FD      1450 STA $FD
CEA8  20FDAE    1470 L004 JSR COMMA
CEAB  2089C0    1480 JSR GETSTR
CEAE  A56C      1490 LDA $6C
CEB0  48        1500 PHA
CEB1  A56B      1510 LDA $6B
CEB3  48        1520 PHA
CEB4  A56A      1530 LDA $6A
CEB6  48        1540 PHA
CEB7  C6FE      1550 DEC $FE
CEB9  D0ED      1560 BNE L004
CEBB  A904      1580 LDA #$4
CEBD  8523      1590 STA $23
CEBF  8526      1600 STA $26
CEC1  8529      1610 STA $29
CEC3  A900      1630 LDA #0
CEC5  8565      1640 STA $65
CEC7  8568      1650 STA $68
```

```
CEC9  856B      1660 STA $6B
CECB  A4FD      1680 LDY $FD
CECD  A200      1700 LDX #0
CECF  68        1710 L005 PLA
CED0  9565      1720 STA $65.X
CED2  68        1730 PLA
CED3  9566      1740 STA $66.X
CED5  68        1750 PLA
CED6  9567      1760 STA $67.X
CED8  E8        1770 INX
CED9  E8        1780 INX
CEDA  E8        1790 INX
CEDB  88        1800 DEY
CEDC  D0F1      1810 BNE L005
CEDE  68        1830 PLA
CEDF  8562      1840 STA $62
CEE1  68        1850 PLA
CEE2  8563      1860 STA $63
CEE4  A903      1880 MLOOP LDA #3
CEE6  85FE      1890 STA $FE
CEE8  A206      1900 LDX #6
CEEA  20E5CD    1910 L006 JSR GETNOT
CEED  F015      1920 BEQ VOCOFF
CEEF  203ECE    1930 JSR CONVER
CEF2  A5FE      1940 LDA $FE
CEF4  20FDCC    1950 JSR DOVOC
CEF7  A514      1960 LDA $14
CEF9  99E002    1970 STA BASE.Y
CEFC  A515      1980 LDA $15
CEFE  99E102    1990 STA BASE+1.Y
CF01  4C11CF    2000 JMP L007
CF04  A5FE      2010 VOCOFF LDA $FE
CF06  20FDCC    2020 JSR DOVOC
CF09  B9E402    2030 LDA BASE+4.Y
CF0C  29FE      2040 AND #$FE
CF0E  99E402    2050 STA BASE+4.Y
CF11  CA        2060 L007 DEX
CF12  CA        2070 DEX
CF13  CA        2080 DEX
CF14  C6FE      2090 DEC $FE
CF16  D0D2      2100 BNE L006
CF18  A018      2120 LDY #$18
CF1A  B9E002    2130 L008 LDA BASE.Y
CF1D  9900D4    2140 STA SID.Y
CF20  88        2150 DEY
CF21  10F7      2160 BPL L008
CF23  A561      2180 LDA $61
CF25  A662      2190 LDX $62
CF27  8563      2200 STA $63
CF29  8664      2210 STX $64
CF2B  0564      2220 ORA $64
CF2D  F013      2230 BEQ NEXTN
```

```
CF2F  20B3EE    2240 L009 JSR DELAY
CF32  A563      2250 LDA $63
CF34  C663      2260 DEC $63
CF36  C900      2270 CMP #0
CF38  D0F5      2280 BNE L009
CF3A  A564      2290 LDA $64
CF3C  C664      2300 DEC $64
CF3E  C900      2310 CMP #0
CF40  D0ED      2320 BNE L009
CF42  A565      2330 NEXTN LDA $65
CF44  0568      2340 ORA $68
CF46  056B      2350 ORA $6B
CF48  D09A      2360 BNE MLOOP
CF4A  AD04D4    2380 OFF LDA SID+4
CF4D  29FE      2390 AND #$FE
CF4F  8D04D4    2400 STA SID+4
CF52  AD0BD4    2410 LDA SID+$B
CF55  29FE      2420 AND #$FE
CF57  8D0BD4    2430 STA SID+$B
CF5A  AD12D4    2440 LDA SID+$12
CF5D  29FE      2450 AND #$FE
CF5F  8D12D4    2460 STA SID+$12
CF62  60        2470 RTS
CF63            2480 END
 TOTAL ERRORS IN FILE --- 0

DELAY           EEB3
BASE            2E0
SID             D400
GETBYT          C006
GETWRD          C000
GETSTR          C089
DOVOC           CCFD
COMMA           AEFD
GETCHR          CDD6
L000            CDE4
GETNOT          CDE5
L001            CDF0
IQERR           CDF8
NOTE            CDFE
EXIT            CE11
BACKUP          CE13
L002            CE19
NONOTE          CE21
OCTAVE          CE24
TABLE           CE37
CONVER          CE3E
CLOOP           CE4C
L003            CE57
NTABLO          CE58
NTABHI          CE64
```

```
PLAY                CE70
L004                CEA8
L005                CECF
MLOOP               CEE4
L006                CEEA
VOCOFF              CF04
L007                CF11
L008                CF1A
L009                CF2F
NEXTN               CF42
OFF                 CF4A
 TOTAL NUMBER OF SYMBOLS --- 36
```

Machine code

ADD	DATA								CHECKSUM
CDD6	B5	65	F0	0A	D6	65	A1	66	9C4C
CDDE	F6	66	D0	02	F6	67	60	20	B8CC
CDE6	D6	CD	F0	37	C9	4F	F0	36	C94A
CDEE	A0	06	D9	37	CE	F0	09	88	7ADA
CDF6	10	F8	20	4A	CF	4C	48	B2	578A
CDFE	C8	C0	06	B0	0E	20	D6	CD	A329
CE06	F0	09	C9	23	D0	07	98	18	9D74
CE0E	69	07	A8	98	60	B5	66	D0	5C30
CE16	02	D6	67	D6	66	F6	65	4C	58DE
CE1E	11	CE	98	A0	00	60	20	D6	5B96
CE26	CD	F0	CF	C9	30	90	CB	C9	CF0F
CE2E	38	B0	C7	29	0F	95	23	90	6712
CE36	AE	43	44	46	47	41	45	42	78A8
CE3E	88	B9	58	CE	85	14	B9	64	906E
CE46	CE	85	15	B5	23	A8	C0	07	9B6F
CE4E	B0	07	46	15	66	14	C8	D0	69B0
CE56	F5	60	1E	8B	06	F3	8F	FA	A504
CE5E	2E	18	7E	AC	E6	F8	86	96	4432
CE66	B3	C8	E1	A8	FD	8E	9F	BD	BE3B
CE6E	D4	EE	20	4A	CF	AD	E4	02	B916
CE76	09	01	8D	E4	02	AD	EB	02	293C
CE7E	09	01	8D	EB	02	AD	F2	02	29BA
CE86	09	01	8D	F2	02	20	06	C0	26DC
CE8E	C9	00	F0	66	C9	04	B0	62	90FA
CE96	85	FE	20	FD	AE	20	00	C0	9C80
CE9E	A5	14	48	A5	15	48	A5	FE	6EE0
CEA6	85	FD	20	FD	AE	20	89	C0	9D52
CEAE	A5	6C	48	A5	6B	48	A5	6A	86FC
CEB6	48	C6	FE	D0	ED	A9	04	85	8CD9
CEBE	23	85	26	85	29	A9	00	85	4441
CEC6	65	85	68	85	6B	A4	FD	A2	7194
CECE	00	63	95	65	68	95	66	68	39B8
CED6	95	67	E8	E8	E8	88	D0	F1	9BB1
CEDE	68	85	62	68	85	63	A9	03	6F09
CEE6	85	FE	A2	06	20	E5	CD	F0	9DBE
CEEE	15	20	3E	CE	A5	FE	20	FD	317D
CEF6	CC	A5	14	99	E0	02	A5	15	A3B7

```
CEFE  99 E1 02 4C 11 CF A5 FE   8FCC
CF06  20 FD CC B9 E4 02 29 FE   7CC8
CF0E  99 E4 02 CA CA CA C6 FE   9E62
CF16  D0 D2 A0 18 B9 E0 02 99   BBE5
CF1E  00 D4 88 10 F7 A5 61 A6   52B4
CF26  62 85 63 86 64 05 64 F0   68E3
CF2E  13 20 B3 EE A5 63 C6 63   3FA3
CF36  C9 00 D0 F5 A5 64 C6 64   9679
CF3E  C9 00 D0 ED A5 65 05 68   947E
CF46  05 6B D0 9A AD 04 D4 29   4829
CF4E  FE 8D 04 D4 AD 0B D4 29   B765
CF56  FE 8D 0B D4 AD 12 D4 29   B861
CF5E  FE 8D 12 D4 60            1698
```

Commentary

Lines 200–310: GETCHR — a subroutine to pick up the next character from the current PLAY string.

Lines 210–230: The length of the string is loaded from a system storage byte and the subroutine returns execution to the main routine if the length is zero.

Line 250: If the length is not now zero, the record of the length still to be processed is decremented by one.

Line 260: The character indicated by the X register in conjunction with the pointer at $66–67 is loaded into the accumulator.

Lines 280–300: The pointer to the next character to be picked up is incremented by one.

Lines 330–920: GETNOT — a subroutine to pick up the next character(s) using the GETCHR subroutine and to translate those characters into note values in the Y register.

Lines 350–360: GETCHR is used to pick up the next character in the string. If the GETCHR routine returns the information that the string is empty, a branch is made to a section of the subroutine which clears the Y register and returns.

Lines 380–390: The character obtained is tested to see if it is 'O'. If so, a branch is made to the section which deals with a change of octave.

Lines 410–450: The character is tested, by means of a comparison with the contents of the table at line 920, against the seven possible character values for notes (a, b, c, d, e, f, g). If it is the same as one of the characters in the table, a branch is made to the section which deals with valid notes.

Lines 470–480: These lines are only executed if the character detected in the string will not produce a valid note. All the voices are turned off and the ILLEGAL QUANTITY error message is generated.

Lines 500–520: At this point a valid note has been found. The next step will be to test to see if the note is to be played as a sharp (ie a semitone higher). Before that, however, a test is made to see if the note is either six or seven, since these two cannot be sharp (the next note up from them in the scale is only a semitone away). If the note *is* six or seven then the lines to detect a sharp will be branched around. Note that if a sharp symbol '#' *is* attached to either note six or seven it will result in an error message the next time through lines 410–480 since the routine will try to interpret the sharp sign as a note value.

Lines 530–540: The next character is picked up from the string and, if the end of the string has been reached, the routine ends.

Lines 560–580: The test for the sharp symbol, ASCII value 35. If what is found is *not* the sharp symbol, the pointer to the place in the string is backed up one to the original character.

Lines 590–620: If the character picked up was a sharp symbol, the note value in the Y register is altered by the addition of seven.

Lines 640–650: The note value is now in the Y register and will be in the range 1–12 (over five means that the note is sharp).

Lines 670–710: The routine called upon earlier to back up the pointer to the position in the string.

Lines 750–770: The lines called when a valid note has not been found. The Y register is cleared, along with the zero flag, before quitting the routine.

Lines 790–800: Earlier (lines 380–390) we noted that a check is made for the 'O' character which indicates that a change of octave is required. These lines begin the section which deals with an octave change by checking that there is actually a character after the 'O', otherwise an error message is generated.

Lines 820–850: A check that the character meant to be specifying the octave is in the range 0–7.

Lines 870–880: Convert the valid octave parameter into the correct value to be stored in the octave register for each voice (see the list of variable

locations). Note that each voice has its own octave register and that the octave for each voice is entirely independent of the other voices.

Line 900: This jump to the beginning of the routine to pick up the next note is not really conditional. At this point in the execution of the routine, having previously tested it in line 850, the carry flag must necessarily be clear so the jump is always made. Using the conditional command simply saves a byte of program memory compared to an absolute jump.

Line 920: The table storing the ASCII values of the allowable note names.

Lines 950–1140: At this point the Y register holds a record of the value of the note to be played and the register for the relevant voice holds the record of the current octave. This subroutine has the task of taking these two values, which are meaningless to the SID chip, and converting them to a frequency value.

Line 950: The note value is at present in the range 1–12 whereas the range needed is 0–11.

Lines 960–990: The two tables at lines 1100–1140 are used to obtain the low and high bytes of the frequency for the particular note. The tables are based on the assumption that the note is in octave seven — if the assumption is incorrect, an adjustment will be made in the subsequent lines. The two-byte frequency is stored temporarily at $14–15.

Lines 1000–1040: Having obtained the frequency value based on the assumption that the octave is seven, the real octave value is loaded into the Y register and then compared with seven. If the actual octave is less than the assumed value of seven, then the frequency value is shifted to the left (divided by two) as many times as the difference between the assumed and the actual octave value. This simple procedure is possible only because any particular note, for instance 'A', in one octave has a frequency twice that of the 'A' in the octave below and half that in the octave above.

Lines 1160–2420: The PLAY command itself. For a full understanding of some of the comments, you might like to look first at the notes on syntax at the end of the section.

Lines 1190–1290: Any voices currently on are switched off. The gate bits, or bits which activate the voices, are then turned on in the pseudo-registers.

Lines 1310–1370: The number of voices specified for PLAY is checked against the maximum of three and stored in $FE.

Lines 1390–1430: The note length required is picked up and stored temporarily on the stack.

Lines 1440–1450: The number of voices is transferred for storage to $FD since $FE is going to be used and corrupted by the following lines.

Lines 1470–1560: Using the COMMA routine and GETSTR, the parameters of the string(s) for the specified number of voices are picked up from the BASIC line and stored on the stack. Thus, although the command line may contain references to up to three strings, one after another, when PLAY begins, the routine will already know exactly where each of the strings in the command is located.

Lines 1580–1610: All the voice octave registers are initially set to four.

Lines 1630–1660: The records of the string length for the three voices are initialised to zero, ie the assumption is made that there is nothing to be played for each voice — a voice will only be turned on later if the corresponding string is found in the command.

Lines 1680–1810: Making use of the spare memory bytes in the floating point accumulators used by the interpreter for arithmetic, the parameters for the PLAY string(s) are recalled from the stack and stored where they will be more accessible to the main part of the routine.

Lines 1830–1860: The note length is pulled from the stack and stored at $62–63.

Lines 1880–2360: The core of the routine, this section controls the picking up of the notes from the PLAY strings and their sounding.

Lines 1880–1900: These lines initialise the variables for the number of voices.

Lines 1910–1920: The first note is obtained by a call to GETNOT. If the equals flag is set on return from GETNOT, this is an indication that the end of the string currently being considered has been reached, so the VOCOFF routine (line 2010) is called to switch off the output for that voice.

Lines 1930–2000: At this point a valid note must have been detected. These lines control its translation into a frequency which is placed into the appropriate pseudo-register for the current voice.

Lines 2010–2050: This section, which is called from line 1920 if the end of the string has been reached, turns off the gate bit in the pseudo-register for the current voice.

Lines 2060–2100: The X register which is used by the subroutines entered earlier to obtain characters from the string(s) by indexed addressing, is decremented by three so that it indicates the variables for the next string. $FE, which was initialised to three at the beginning of the main loop (line 1880), is decremented by one and the main loop executed again if all three voices have not been dealt with.

Lines 2120–2160: By this point, the pseudo-registers for the voices have been set up in such a way as either to turn the voice(s) off or to play a note. These lines perform the task of transferring the contents of the pseudo-registers to the actual SID registers. As soon as this is done, of course, the notes dictated by the contents of the pseudo-registers will be played by the SID.

Lines 2180–2210: The note length is moved to $63–64 where it can be used as a counter and its value can be decremented.

Lines 2220–2230: These two lines effectively test whether the note length was zero by comparing the high byte of the value with the low byte. If the two are equal to zero then a branch is made to the section which calls up the next note.

Lines 2240–2320: The two-byte value contained in $63–64 is decremented by one for each iteration of this loop until zero is reached. In this way the ending of the note is delayed and the note timed.

Lines 2330–2360: The series of OR instructions test whether any of the PLAY strings have not been exhausted according to the variables which record the remaining length for each of the string(s). If there is any part of any of the strings left unprocessed then the main loop, beginning at line 1880, is executed again to pick up the next note.

Lines 2380–2410: These lines come into effect when all the PLAY strings have been exhausted and have the effect of switching off all the voices. Note that this operation is carried directly on the SID and not on the pseudo-registers.

Testing

PLAY is an extremely complex command which it is difficult to test fully in any quick way. The best way to approach testing is to try to examine its functions one by one.

Having loaded all the sound commands with the exception of BEEP and CHORD, start with the following small BASIC program:

```
20 FOR I = 1 TO 3
30 SYS (52532) I,0,15,0,0 : REM ADSR
40 SYS (52635) I,0,16 : REM VOICE CONTROL – FILTER OFF,
   SAWTOOTH WAVE
50 NEXT
60 SYS (52498) 15 : REM SET UP VOLUME
70 SYS (52848) 1,500,"CDEFGABO5C"
80 END
```

The effect of running the program should be to play a scale of C — provided of course that you have remembered to turn up the TV speaker. If all is well so far, change line 70 to read:

```
70 SYS (51824) 1,500,"CC # DD # EFF # GG # AA # B"
```

You should now find that the program plays the whole of the scale of C for octave four, including semitones. Again, if all is well, you can modify the BASIC program further as follows:

```
65 FOR K = 2 TO 7
70 SYS (51824) 1,500,"O" + CHR$(48 + K) + "CC # DD # EFF # GG #
AA # B"
75 NEXT K
```

This should play, in ascending order, all of the notes which you are likely to be able to hear through your TV speaker, since the first two octaves are too low to be heard effectively.

Syntax and notes on use

The syntax for PLAY is:

SYS (51824) { VOICES} ,{ NOTE LENGTH} ,{ PLAY STRING 1}
[,{ PLAY STRING 2} ,{ PLAY STRING 3}]

The use of VOICES and NOTE LENGTH is very much as in CHORD, except that PLAY is intended to deal with a series of chords rather than a single one. The PLAY strings need not be of the same length and may contain any of the following:

1) The letters A, B, C, D, E, F, G to represent individual notes.
2) The 'sharp' symbol (#) following the letter for a note, indicating that the note is to be raised by one semitone.
3) The letter 'O', followed by a number in the range 0–7, to indicate the octave from which the notes following are to be taken.

Since PLAY limits each voice to notes of the same length for the duration of the string to be PLAYed, in practice we have found it more practical to include the play strings in DATA statements so that the note length and play strings can be READ by some kind of loop and then be PLAYed. In this way, note length can be varied in the course of playing without having constantly to enter new PLAY commands. In this case the PLAY command might take the form of:

PLAY 3, NL, P1$,P2$,P3$

with the note length and strings being read from DATA statements.

An example of this technique is given in the program below:

```
1 GOTO3
2 SAVE"@0:PLAY TEST",8:VERIFY"PLAY TEST"
,8:STOP
3 REM
5 NV = 1
10 REM PROGRAM TO TEST PLAY COMMAND
15 IF NV=3 THEN READA$ : IF A$<>"-1" THE
N 15
20 FOR I =  1 TO 3
30 SYS (52532) I,8,9,2,5 : REM  ASDR
40 SYS (52635) I,0,32 : REM VOICE CONTRO
L
50 NEXT
60 SYS (52498) 15 : REM SET UP VOLUME
70 READ NL : IF NL=-1 THEN END
80 READ T1$
85 T3$="" : T2$ = "" : IF NV=3 THEN READ
 T2$,T3$
90 SYS (52848) 3,NL,T1$,T2$,T3$
100 GOTO 70
500 DATA500,05C,500,05CD,750,04B,250,05C
,500,05DEEF,750,05E,250,05D,500,05CDC04B
510 DATA 750,05C
520 DATA -1
600 DATA 500,05CCD,06CCD,04CCD
610 DATA 750,04B,05B,03B
700 DATA -1

READY.
```

CHAPTER 6
The BASIC Extender

In a sense, this chapter is what the whole of this book has been about. We hope that you have enjoyed entering and playing with the routines in the book. The promise, however, was that the contents of the book would eventually be more than a host of machine code to be called by interminable SYS commands.

In this chapter, we shall show you how to make the routines you have so far entered fully part of the BASIC language, each with its own keyword. Of course, they will not become part of the permanent memory of the 64, but you will nevertheless be able to write programs using them and, provided you remember to load the machine code first, recall those programs at any time and RUN them. The keywords will list normally on the screen, they will output to a printer just like any others and they will also do everything that the SYS commands earlier in the book could achieve.

We should make it clear here that we do not intend to give a full commentary on the BASIC Extender. This is not because it is such a complex program in terms of the BASIC language but because it is designed to make small changes to extensive routines in the BASIC interpreter of the 64. To explain every detail of the Extender would therefore involve listing out the ROM routines and giving chapters of commentary on *them* before turning to the Extender itself.

If, after using the Extender, you feel you would like to go further into the question of modifying BASIC, you will find that our previous book, *Commodore 64 Machine Code Master*, goes into more detail on the methods involved. Do not ignore the fact, however, that the present Extender program is designed specifically so that you can add your own new keywords, together with the start address of the routine to execute them, and have them automatically added to BASIC.

In brief, what we shall be doing with the Extender is to alter three aspects of the normal working of the system — the crunch tokens, print tokens and execute keywords routines. Keywords within BASIC programs are not stored in their full form but in the form of tokens, or single characters with values in the range 128–202. When a keyword is entered in a new line a section of the ROM is employed to transform it into such a token. When adding new keywords to BASIC, it will be necessary to alter the execution

of this ROM routine since it will otherwise not recognise our new commands.

The reverse of this 'crunching' process must take place when a program is listed, since the keyword tokens must be recognised in order that they may be retranslated into their full form. Finally, the ROM routine which executes each BASIC command on the basis of its token value must be persuaded to recognise the extra tokens required for our new keywords.

Basic Extender

```
3 REM
9000 REM********************************
9001 REM BASIC EXTENDER
9002 REM********************************
9010 DEF FNHI(X) = INT(X/256)
9020 DEF FNLO(X) = X-256*FNHI(X)
9030 CSTART = 49799 : CCODE = 42364 : CL
 = 150 : COFF= 9
9040 PCODE = 42789 : PL = 28 : PSTART =
53120 : POFF = 20
9050 REM EXTRA IS SET UP IN ROUTINE 9700
 - 9780
9060 ECODE = 42999 : EL = 12 : EOFF = 28
 : ESTART = PSTART+POFF+PL+4
9070 VTABLE = 50691
9090 NTABLE = CSTART+COFF+CL+7 : NKEY =
75 : EXTRA = 0
9100 REM********************************
9101 REM CRUNCH TOKENS
9102 REM********************************
9110 FOR I = 0 TO CL : POKE I+COFF+CSTAR
T,PEEK(I+CCODE) : NEXT
9120 POKE CSTART  ,32  : POKE CSTART+1,1
24 : POKE CSTART+2,165
9130 POKE CSTART+3,230 : POKE CSTART+4,1
22
9140 POKE CSTART+5,208 : POKE CSTART+6,2
9150 POKE CSTART+7,230 : POKE CSTART+8,1
23
9160 POKE CSTART+COFF+65 ,FNLO(NTABLE  )
 : POKE CSTART+COFF+66 ,FNHI(NTABLE  )
9170 POKE CSTART+COFF+127,FNLO(NTABLE-1)
 : POKE CSTART+COFF+128,FNHI(NTABLE-1)
9180 POKE CSTART+COFF+132,FNLO(NTABLE  )
 : POKE CSTART+COFF+133,FNHI(NTABLE  )
9190 POKE CSTART+20,48 : POKE CSTART+21,
64
9200 POKE CSTART+COFF+52,240 : POKE CSTA
RT+COFF+53,97
9210 POKE CSTART+COFF+151,169 : POKE CST
ART+COFF+152,NKEY
```

```
9220 POKE CSTART+COFF+153,133 : POKE CST
ART+COFF+154,11
9230 POKE CSTART+COFF+155,208 : POKE CST
ART+COFF+156,153
9240 POKE 772,FNLO(CSTART) : POKE 773,FN
HI(CSTART)
9300 REM*********************************
9301 REM PRINT TOKENS ROUTINE
9302 REM*********************************
9310 FOR I = 0 TO PL : POKE PSTART+I+POF
F,PEEK(PCODE+I) : NEXT
9320 POKE PSTART  ,16  : POKE PSTART+1,8
9330 POKE PSTART+2,201 : POKE PSTART+3,2
55
9340 POKE PSTART+4,240 : POKE PSTART+5,4
9350 POKE PSTART+6,36  : POKE PSTART+7,1
5
9360 POKE PSTART+8,16  : POKE PSTART+9,3
9370 POKE PSTART+10,76 : POKE PSTART+11,
243 : POKE PSTART+12,166
9380 POKE PSTART+13,201 : POKE PSTART+14
,NKEY+128
9390 POKE PSTART+15,176 : POKE PSTART+16
,3
9400 POKE PSTART+17,76  : POKE PSTART+18
,36  : POKE PSTART+19,167
9410 POKE PSTART+POFF+12,FNLO(NTABLE) :
POKE PSTART+POFF+13,FNHI(NTABLE)
9420 POKE PSTART+POFF+20,FNLO(NTABLE) :
POKE PSTART+POFF+21,FNHI(NTABLE)
9430 POKE PSTART+POFF+1,NKEY+127
9440 POKE PSTART+POFF+23,5
9450 POKE PSTART+POFF+29,76 : POKE PSTAR
T+POFF+30,239 : POKE PSTART+POFF+31,166
9460 POKE 774,FNLO(PSTART) : POKE 775,FN
HI(PSTART)
9500 REM*********************************
9501 REM SET UP TABLES
9502 REM*********************************
9510 RESTORE : ADDRESS = NTABLE
9520 READ A$ : IF A$<>"FOR TABLES" THEN
 9520
9530 READ A$
9540 IF A$="END" THEN 9590
9550 FOR I = 1 TO LEN(A$)
9560 POKE AD+I-1,ASC(MID$(A$,I,1)) AND 1
27
9570 NEXT
9580 AD = AD+LEN(A$) : READ A$ : POKE AD
-1,PEEK(AD-1) OR 128 : GOTO 9530
9590 POKE AD,0 : AD = AD+1
9600 REM*********************************
```

```
9601 DATA FOR TABLES
9602 REM*******************************
9610 DATA BANK,C0CA
9611 DATA LOCATE,C10A
9612 DATA SIZE,C11C
9613 DATA SCREEN,C147
9614 DATA CHRPTR,C180
9615 DATA COLOUR,C1B0
9616 DATA PLOT,C1C2
9617 DATA ALLOT,C24C
9618 DATA CCOPY,C013
9620 REM ***** HIRES ******
9621 DATA HIRES,C73B
9622 DATA LOW,C700
9623 DATA HSCREEN,C764
9624 DATA HLOCATE,C81E
9625 DATA HPLOT,C868
9626 DATA LINE,C9CF
9627 DATA CIRCLE,CB14
9628 DATA TEXT,CBDC
9630 REM ***** SPRITES *****
9632 DATA SCOLOUR,C474
9633 DATA ENLARGE,C48A
9634 DATA SPTR,C4BE
9635 DATA DESPRITE,C4EF
9636 DATA RESPRITE,C50B
9637 DATA SHAPE,C5C3
9638 DATA SPRITE,C417
9640 REM ***** SOUND *****
9641 DATA BEEPS,CC5F
9642 DATA BEEP,CC00
9643 DATA VOLUME,CD12
9644 DATA ADSR,CD34
9645 DATA FILTER,CD57
9646 DATA VOICE,CD9B
9647 DATA PLAY,CE70
9648 DATA PULSE,CF64
9650 DATA END
9700 REM*******************************
9701 REM SET UP THE VECTOR KEYTABLE
9702 REM*******************************
9710 RESTORE : EXTRA = 0 : AD = VTABLE
9720 READ A$ : IF A$<>"FOR TABLES" THEN
 9720
9730 READ A$ : IF A$="END" THEN 9800
9740 READ A$ : EXTRA = EXTRA+1
9750 T = 0
9760 FOR I = 1 TO LEN(A$)
9770 T1 = ASC(MID$(A$,I,1))-48 : IF T1>-
1 THEN T = T*16+T1+(T1>9)*7
9780 NEXT : POKE AD,FNLO(T-1) : POKE AD+
1,FNHI(T-1) : AD = AD+2 : GOTO 9730
```

```
9800 REM*********************************
9801 REM EXECUTE KEYWORD ROUTINE
9802 REM*********************************
9810 POKE ESTART   ,32 : POKE ESTART+1,F
NLO(ES+13) : POKE ESTART+2,FNHI(ES+13)
9820 POKE ESTART+3 ,76 : POKE ESTART+4 ,
174 : POKE ESTART+5 ,167
9830 POKE ESTART+6 ,76 : POKE ESTART+7 ,
43 : POKE ESTART+8 ,168
9840 POKE ES+9,56 : POKE ESTART+10,76 :
POKE ESTART+11,239 : POKE ESTART+12,167
9850 POKE ESTART+13,32 : POKE ESTART+14,
115 : POKE ESTART+15,0
9860 POKE ESTART+16,240 : POKE ESTART+17
,244
9870 POKE ESTART+18,201 : POKE ESTART+19
,NKEY+128
9880 POKE ESTART+20,144 : POKE ESTART+21
,243
9890 POKE ESTART+22,201 : POKE ESTART+23
,EXTRA+NKEY+128
9900 POKE ESTART+24,176 : POKE ESTART+25
,239
9905 POKE ESTART+26,233 : POKE ESTART+27
,NKEY+127
9910 FOR I = 0 TO EL                    :
9920 POKE I+ESTART+EOFF,PEEK(I+ECODE)
9930 NEXT
9940 POKE ESTART+EOFF+3,FNLO(VTABLE+1) :
 POKE ESTART+EOFF+4,FNHI(VTABLE+1)
9950 POKE ESTART+EOFF+7,FNLO(VTABLE) : P
OKE ESTART+EOFF+8,FNHI(VTABLE)
9960 POKE 828,169 : POKE 829,FNLO(ESTART
) : POKE 830,141 : POKE 831,8
9970 POKE 832,3 : POKE 833,169 : POKE 83
4,FNHI(ESTART) : POKE 835,141
9980 POKE 836,9 : POKE 837,3 : POKE 838,
96
9990 SYS 828
```

Commentary

Lines 9000–9090: This module sets up the variables for the program, including the present addresses of the ROM crunch tokens (CCODE), print tokens (PCODE) and execute (ECODE) routines. The variables ending in 'START' represent the new addresses at which the routines will be placed and the variables ending in 'L' are the length of each routine. Each routine has an offset, or gap between the specified new start address and the actual start of the code, the length of the gap being represented by variables ending in 'OFF'. These gaps are to allow for the insertion of extra

coding needed to interface the new commands. NTABLE is the start address of the new keyword table we shall create.

Lines 9100–9240: This module deals with the necessary changes to the crunch tokens routine. The routine is copied into RAM at the new address specified and then the interface routine is added to the beginning. Various pointers are changed including the address at which CRUNCH TOKENS will look for the keyword table. Changes are made to the execution of the routine, to stop it deleting the tokens for our new keywords when it first scans the line looking for the normal keywords. Finally, the vector which points to CRUNCH TOKENS is altered.

Lines 9300–9460: These lines copy the PRINT TOKENS routine, then add the interface routine and the address of the new keyword table. The only change made to the routine as it was in the ROM is that a branch instruction, to a byte which is too far away to be reached by branch once PRINT TOKENS has been moved, is changed so that it points to a jump instruction which *can* span the gap.

Lines 9500–9590: These lines pick up the new keywords from the list in the next section and load them into the new keyword table. The code for the last letter of each keyword is ANDed with 128, the normal method of marking the end of a keyword in the table.

Lines 9600–9650: The new keywords and the hex address at which the corresponding routines start.

Lines 9700–9790: With a second scan through the table in the previous section, the start addresses of the new routines are loaded into the new vector table.

Lines 9800–9990: These set up the interface routine for the EXECUTE KEYWORD then copy the ROM routine to RAM. The address of the new vector table is POKEd into the new routine. Finally, a small machine code program is entered which will alter the value of the pointer used by the system to jump to EXECUTE KEYWORDS. This cannot be done in BASIC since the pointer consists of two bytes. Having changed one of the bytes, the next POKE command would lead to a crash since the system would attempt to find EXECUTE KEYWORDS using the half-altered pointer.

Testing

To test the functioning of this program it is wiser not to jump in the deep end by trying to make it point immediately to all the new commands. Errors will be very difficult to detect.

Probably the best test, having SAVEd the program, is to alter all the addresses in the DATA statements between lines 9600 and 9650 to read $A871 then *save the program again* under a different name. $A871 is in fact the address of the ROM routine to execute RUN. Having made the change,

RUN the program and, if it finishes without flagging any BASIC errors, delete it with NEW.

Now enter a new program consisting of the following single line:

```
10 PRINT "COMMAND O.K."
```

Having done that, you can begin to work through all the keywords in the Extender listing, entering them in direct mode (ie directly on to the screen without a line number). If the Extender has worked correctly, when you press RETURN after typing the keyword, you should see 'COMMAND O.K.' appear underneath it. The reason for this is that every one of the keywords is simply executing RUN.

Remember that you are only entering the keywords which appear in lines 9610–9647 of the Extender. There are many routines in the machine code which are not commands to be called from BASIC. Entering their names would simply create a confusing SYNTAX ERROR message.

If all the keywords check out, then you are ready to go back to the Hex Loader (or whatever means you have used for entering the machine code commands) for a moment, to load into memory a sizeable chunk of machine code like the low resolution commands section (including the parameters routines, of course). Having placed the machine code into memory, do a brief test of one or two of the commands to ensure that the machine code side of things is working correctly. Now load the first listing of the Extender, the one with the correct addresses for all the machine code routines. Double-check the addresses specified, they are now all that stands between you and final success. When you are sure that the addresses are exactly as listed above, you are ready to RUN the program again.

When the Extender has finished its work, you should be able to enter any of the new keywords and find that they perform exactly as the original SYS commands, with one important exception. Though the BASIC Extender modifies the execution routine of the BASIC interpreter so that it will recognise the new keywords, no action is taken to ensure that they will be recognised after IF...THEN, so that a line like:

```
10 IF X> 10 THEN BEEP 500,10000
```

would produce a syntax error. Don't despair, this doesn't drive a coach and horses through the promise to integrate the new commands with BASIC. All that you need to do is to place a colon (:) after the THEN and before the new keyword, eg:

```
10 IF X> 10 THEN : BEEP 500,1000
```

This is perfectly legal in BASIC and makes no difference to the operation of the IF...THEN statement. What it *does* do is to persuade the BASIC interpreter that the keyword falls at the beginning of a line or a statement within a line, thus ensuring that it is recognised.

To make doubly sure that everything is working correctly, the following short BASIC program uses every one of the new commands:

```
1 GOTO 3
2 SAVE "@0:TEST LISTING",8 : VERIFY "TES
T LISTING",8 : STOP
3 REM
4 PRINT"[CLR]":COLOUR1,1:LOCATE20,10:PRI
NT"[GREEN]THIS IS A BEEP":BEEP 1000,50
6 T=PEEK(43)+256*PEEK(44) : IF T-2049>=1
024 THEN GOTO 20
7 ALLOT 1024
20 RESPRITE : RESTORE
25 PRINT "[CLR]" : COLOUR 5,5 : LOCATE 2
0,10:PRINT "[WHITE]THIS IS A SPRITE "
30 SP$  = "         **********         "
40 SP$  = "       **************       "
50 SP$  = "      ****************      "
60 SP$  = "     ******************     "
70 SP$  = "    ********************    "
80 SP$  = "    ********************    "
90 SP$  = "    ********************    "
100 SP$ = "    ********************    "
110 SP$ = "    ********************    "
120 SP$ = "     ******************     "
130 SP$ = "      ****************      "
140 SP$ = "       ************         "
150 SP$ = "       **   ***   **        "
160 SP$ = "      **    ***    **       "
170 SP$ = "     **   *******   **      "
180 SP$ = "    **    *******    **     "
190 SP$ = "  **                    **   "
200 SP$ = "******              ****** "
210 SP$ = "******              ****** "
220 SP$ = "                           "
221 SP$ = "                           "
222 SP$ = "                           "
225 SPTR 0,832 : SHAPE 832
240 SCOLOUR 0,1
250 ENLARGE 0,2,2
270 FOR I = 20 TO 340 STEP 2
280 SPRITE 0,I,100
295 BEEPS 2,50,7000,1000
300 NEXT I
310 DESPRITE
315 PRINT "[CLR]" : COLOUR 6,6
320 ADSR 1,0,9,0,9 : ADSR 2,5,8,5,9
325 VOLUME 15
330 VOICE 1,0,16 : VOICE 2,0,64
335 PULSE 2,2000
340 READ NL: IF NL<0 THEN 390
350 READ T1$,T2$
```

```
360 PLAY 2,NL,T1$,T2$
370 DATA 500,CC#DD#EFF#AA#B,O7CC#DD#EFF#
AA#B
390 HSCREEN 8192,2048
400 HIRES : TEXT "[CLR]"
410 HLOCATE 90,100
420 TEXT "THIS IS HIRES TEXT"
430 LINE 0,50,50,270,150
440 CIRCLE 0,80,160,100
450 FOR I = 0 TO 1000
460 HPLOT 0,RND(1)*120+100,RND(1)*60+80
470 NEXT
510 FOR I = 0 TO 5000 : NEXT
520 LOW
530 CCOPY 18
540 BANK 1
580 CHRPTR 18
590 COLOUR 0,5
600 PRINT "[WHITE]"
610 SIZE 24,38 : PRINT "[CLR]"
620 FOR MODE = 0 TO 2
630 FOR I = 20 TO 60+(MODE=1)*20
640 PLOT MODE,20,I
650 NEXT I,MODE
660 LOCATE 20,10 : PRINT "THIS IS LOW RE
S TEXT"
690 FOR I = 0 TO 5000 : NEXT
700 BANK 0
710 CHRPTR 6
720 SIZE 25,40
730 PRINT "[CLR]"
```

Tidying things up

At this stage, it is quite possible that you have a mass of different versions of the Hex Loader, plus the BASIC Extender floating around. The final stage is to make up a single package which can be loaded from tape or disk and which contains all the machine code and the Extender. Provided that you have not made any changes (other than adding different DATA lines) to different versions of the Hex Loader, and also provided that *none of the lines containing DATA have the same numbers in the different versions*, the procedure is as follows:

1) Make sure you have one long tape or a number of shorter ones available.
2) Load each of the different versions of the Hex Loader into the 64, have a tape ready in the cassette recorder and then enter:

   OPEN 1,1,2,"MERGE":CMD1:LIST

when the cassette recorder finally stops and the screen returns to you, enter:

```
PRINT#1 : CLOSE 1
```

which completes the listing to the tape.

3) When you have finished storing all the versions of the Hex Loader in this way, do the same for the BASIC Extender.
4) Enter the Merge program you will find in Appendix B.
5) Run the Merge program and you will be prompted to switch on the tape recorder. Replay any of the programs you have stored and wish to merge. When one has finished, enter RUN 63990 to start the Merge program again and work on the next program to be merged in. In the process, any lines with the same number as those being loaded from tape will be overwritten, so unless they are the same in content, something will be lost.
6) At the end of all this you will have a program consisting of the Hex Loader and *all* the DATA lines required for the complete set of commands, plus the BASIC Extender itself. RUNning the complete program will result (slowly) in the new commands being entered into memory and patched into BASIC as keywords.
7) One further step will speed things up considerably. Once all the commands are in memory, regardless of whether or not the BASIC Extender has been RUN, clear out the existing BASIC program with NEW and then enter the following short BASIC program:

```
10 POKE 43,0 : POKE 44,192 : POKE 45,0 : POKE 46,208
20 SAVE "EXTENDED BASIC",1
```

(For disk drive owners the final '1' in line 20 should be replaced with an '8'.) RUN the program, and the 4K area of memory from 49152 onwards will be saved as if it were a program. To retrieve the code in future, all you now need to do is to enter:

```
LOAD "EXTENDED BASIC",1,1
```

(LOAD "EXTENDED BASIC",8,1 if you are using a disk drive.)

and the machine code will be loaded back into memory in the place from which it was originally taken. When this has been done, enter NEW, then LOAD and RUN the BASIC Extender. The time saving is considerable since even from cassette the machine code will reload in around 90 seconds.

APPENDIX A
Hex Loader

All the routines in this book were written and assembled on the Mastercode Assembler which began life in our first book, *Commodore 64 Machine Code Master*, and is now available in commercial form from Sunshine. By far the most practical way to approach this book, or any other substantial piece of machine code, is to use such an assembler. We recognise, however, that not everyone will be working with an assembler and all the routines in the book are accompanied by tables of hex values which can be entered using the Hex Loader described in this section. The program will accept hexadecimal information in the form of data statements, format a table for comparison with what is in the book, and even provide a checksum for each group of eight values which you can use to ensure that the data has been entered correctly.

Hex Loader Listing

```
HEX LOADER - HEX LOADER
1 GOTO3
2 SAVE "@0:HEX LOADER",8 : VERIFY "HEX L
OADER",8 :END
3 REM
5 ZZ=0
10 REM**********************************
11 REM HEX LOADER FOR C64
12 REM**********************************
20 DEFFN HEX(X) = X-48+(X>64)*7+(X>57 AN
D X<65)*100
25 POKE 53280,13 : POKE 53281,5
30 PRINT "[CLR][CD]" SPC(10) "HEX LOADER
"
40 AD = 49152
60 INPUT "[CD]PUT IN MEMORY (Y/N) ? N[CL
][CL][CL]";T$
70 PM = T$="Y" : IF (NOT PM) AND T$<>"N"
 THEN PRINT "[CU][CU][CU]" : GOTO 60
80 RESTORE : GOSUB 1120 : CK = 0
90 READ HE$
```

```
100 IF HE$="END OF CODE" THEN GOSUB 1500
 : GOSUB 1200 : PRINT "[CLR]" : END
110 IF LEFT$(HE$,1)<>"0" THEN 150
115 HE$ = MID$(HE$,2)
120 GOSUB 1000
130 IF ERR THEN 1400
140 AD = HE : IF ZZ THEN GOSUB 1500 : GO
SUB 1100
145 GOTO 90
150 GOSUB 1000
151 IF ERR THEN 1400
158 IF PM THEN POKE AD,HE
170 PRINT RIGHT$("00"+HE$,2) " " ; : CC
= CC+1 : AD = AD+1 : CK = 2*CK+HE
175 ZZ=-1
180 IF CC>7 THEN GOSUB 1500 : GOSUB 1140
190 GOTO 90
1000 REM*********************************
1001 REM CONVERT HEX IN HE$ TO 2 BYTE
1002 REM*********************************
1010 HE = 0 : ERR = 0
1020 FOR I = 1 TO LEN(HE$)
1030 T = FNHEX(ASC(MID$(HE$,I,1)))
1040 ERR = ERR OR T<0 OR T>15
1050 HE = HE*16+T
1060 NEXT I
1070 ERR = ERR OR HE<0 OR HE>65535
1080 RETURN
1100 REM*********************************
1101 REM PRINT PAGE HEADINGS
1102 REM*********************************
1110 GOSUB 1200
1120 CL = 3
1130 PRINT "[CLR][CD]ADD   DATA
            CHECKSUM[CD]"
1140 T = AD : GOSUB 1300
1145 CC = 0 : CL = CL+1 : IF CL>20 THEN
1100
1146 PRINT "" RIGHT$("0000"+T$,4) "  " ;
1150 RETURN
1200 REM*********************************
1201 REM WAIT FOR KEY PRESS
1202 REM*********************************
1220 IF CL<22 THEN CL = CL+1 : PRINT : G
OTO 1220
1270 PRINT "     PRESS A KEY TO CONTINUE
 ..."
1280 GET T$ : IF T$="" THEN 1280
1290 RETURN
1300 REM*********************************
1301 REM CONVERT DECIMAL IN T TO HEX T$
```

```
1302 REM*********************************
1310 T$ = "" : T2 = 4096
1315 FOR I = 0 TO 3
1320 T1 = INT(T/T2)
1330 T = T-T1*T2
1340 T2 = T2/16
1350 T$ = T$+CHR$(T1+48-(T1>9)*7)
1360 NEXT
1370 RETURN
1400 REM*********************************
1401 REM DEAL WITH ERRORS IN DATA
1402 REM*********************************
1410 PRINT "[CLR][CD][CD][CD]"
1415 LN = PEEK(63)+256*PEEK(64)
1420 PRINT "  ********   ERROR IN FILE *
********"
1430 PRINT "[CD]"
1440 PRINT "     IN LINE NUMBER :-" LN
1450 PRINT "[CD][CD][CD][CD][CD]"
1460 PRINT "LIST" LN  "[CU][CU][CU]"
1470 POKE 198,1
1480 POKE 631,13
1490 END
1500 REM*********************************
1501 REM PRINT THE CHECKSUM
1502 REM*********************************
1510 T = CK-INT(CK/65536)*65536
1520 CK = 0
1530 GOSUB 1300
1540 PRINT ""   SPC(2+(8-CC)*3) T$
1550 RETURN
10000 REM********************************
10001 REM MACHINE CODE DATA
10002 REM********************************
63800 DATA END OF CODE
63900 REM MERGE ROUTINE
63910 OPEN 8,8,8,"MACHINE CODE,S"
63920 POKE 184,8 : POKE 185,104 : POKE 1
86,8 : POKE 152,1
```

Commentary

This is not intended to be a full commentary on the program, merely a brief indication of the way the program works and its use.

Line 20: This function translates a valid hexadecimal character in the range 0–F into a decimal value.

Line 60: The program provides the user with the option of either simply producing a table so that the data can be checked, or of producing a table and placing the data simultaneously into memory.

Line 100: When hexadecimal is placed into DATA statements for translation, it should be terminated by a data line reading END OF CODE.

Line 110: Whenever the Hex Loader comes across a hex value preceded by the letter 'O', it takes the value to be an address. Subsequent hex values will be compiled from that address onwards. The program may contain more than one 'O' address, so that several sections of code, beginning at entirely different places, can be entered at the same time.

Lines 170–180: The table of hexadecimal values will be printed out in lines of eight.

Lines 1000–1080: This section uses the function defined at the beginning of the program to translate a two-character hex value into decimal. An error is generated if a character is encountered which is outside the two ranges 0–9 and A–F, or if the resultant value is outside the range 0–65535.

Lines 1100–1150: This section prints the heading to the table of hex values. It also prints the memory address at which data will be placed at the beginning of each line of eight hex values.

Lines 1200–1290: This section produces a pause at the end of each page if more data is to be processed. The pause will continue until a key is pressed.

Lines 1300–1370: This section translates decimal values, such as the address for data, into a string representing the value.

Lines 1400–1490: This section flags any errors in the entry of the hex data, such as incorrect format. The system pointer which records which line of DATA is being READ is used to pinpoint the position of the error.

Lines 1500–1550: This section generates the checksums which appear at the end of each line of eight values. The checksum works by adding the next value in the line and then doubling each time. This indicates the presence both of incorrect characters and of data items whose positions have been reversed.

Lines 10000–end: Hex data is placed in DATA statements in this section, terminated with line 63900.

Testing

Turn to the section on the machine code routine GETWRD. This is a short routine so takes little or no time to enter. Look at the table for GETWRD.

The first step is to enter one line of DATA READing "DATA OC000". This is the start address at which the hex will eventually be placed. For each routine that you subsequently enter, the start position will be the first address on the hex table for that routine. Now enter further DATA statements into the Hex Loader with each line of eight values in the table representing one line of DATA. Where the last line of the table has less than eight values, simply enter the values shown. *Do not* enter the checksum at the end of each line in the table as an item of DATA.

When all the values have been entered, RUN the program, specifying the table creation without the data being placed into memory. The result should be a table (of machine code listings) which corresponds exactly to the table in the book, including the checksums. Provided that all is well, you can now go on and use the Hex Loader to place the data into the memory.

Notes on use

When you have finished entering and checking the DATA, do not simply erase the program. Save it under the name of the machine code routine it contains and you will be able to recall it, with the data, rather than having to enter the data again. As mentioned in the commentary, using the 'O' specifier to show that a new address is being used will allow you to enter several routines, one after another, in the same Hex Loader program. If you wish, you may enter all of the routines in this book into one program. To begin with, however, we would recommend that you make up one copy of the Hex Loader including all of the first four parameter routines from Chapter 1. From this, you can make up four separate programs for low resolution, high resolution, sprite and sound commands. Each time you want to enter a new command you need only call up the version of the Hex Loader which deals with that section of the book, enter the start address of the new routine in the DATA line, preceded by 'O', and then enter what you see in the hex table for the routine.

Note: Each of these versions of the Hex Loader should have one section of DATA statements which are exactly the same, ie the parameters routines from Chapter 1 and another section of DATA statements with line numbers which are unique to that version. The line numbers for the DATA statements which create the high resolution commands, for instance, should be completely different to those in the version of the Hex Loader which creates the low resolution commands. The reason for this is that you may later wish to merge all your versions of the Hex Loader together using the method described in Chapter 6 and, in doing so, a line in memory with the same number as a line being merged will be overwritten.

APPENDIX B
Merge Program and Control Characters

The program given below is the Merge routine whose use is described in Chapter 6. It is taken from *The Working Commodore 64* by David Lawrence, where you will find a full explanation of its working.

```
63990 PRINT "[CLR]" : REM HEART SYMBOL ON CLR/HOME
      KEY
63991 OPEN 1,1,0, "COMMANDS"
63992 POKE 184,1:POKE 185,96:POKE 186,1:POKE 152,1:PRINT
      "[CLR] [CURSOR DOWN]"
63994 GET # 1,A$:PRINT A$;:IF ST THEN 63999
63995 IF A$< > CHR$(13) THEN 63994
63996 PRINT "GOTO 63992[HOME]";:POKE 631,13:POKE
      632,13:POKE 633,13
63997 POKE 198,3:END
63999 CLOSE 1
```

To avoid the difficulties in reproducing control characters such as cursor controls in listings, all control characters have been replaced in the BASIC listings by abbreviations placed in square brackets. The control characters and the form in which they may appear in listings are given below:

CURSOR UP	[CU]
CURSOR DOWN	[CD]
CURSOR LEFT	[CL]
CURSOR RIGHT	[CR]
CLEAR SCREEN	[CLR]
HOME CURSOR	[HOME]

Colour control characters are represented by the name of the colour within square brackets, eg [WHITE].

>Here's our way of saying Thanks for buying this Sunshine book...

>Sunshine Special Offer 1

Subscribe to Commodore Horizons, Britain's brightest Commodore magazine, and get an extra three months' issues absolutely free!

For just £10 you get a year's subscription plus the next 3 months without charge – a saving of £2.50. Complete and send the order card now, to reserve your special subscription, along with your cheque or postal order, or charge to your Access card.

See over for two other free Sunshine Offers.

>See over for two other free Sunshine Offers ➡

>Sunshine Special Offers Claim Card

Yes, I'd like to take advantage of your Sunshine Special Offers.
Please make sure I get the following:

☐ A year's subscription to Commodore Horizons, plus 3 months free issues, for just £10.

☐ A year's subscription to Popular Computing Weekly, plus 12 issues free, for just £19.95.

☐ A copy of your free colour Sunshine Books brochure.

Payment methods:

☐ I enclose a cheque/postal order payable to Sunshine Books for £_____

☐ Charge my Access card No._____

Signed_____

Name_____

Address_____

_____

All offers end on 31st December 1984.

>Sunshine Special Offer

2

Popular Computing Weekly is the best-selling weekly newsmagazine for every computer whiz. And here's your opportunity to get 12 issues, worth £4.20, free of charge.

Use the special offer order card to claim your special Popular Computing Weekly subscription – 64 issues for the price of a year's copies – at a price of just £19.95. Send today, and look forward to 15 months of great weekly reading.

>Sunshine Special Offer

3

If you liked this Sunshine Book, take a look at the rest of our amazing range. Send today for our free colour brochure detailing all the Sunshine Books you can buy today, and all the new books on the way.

And once you're on our special mailing list, we'll keep you updated on what's new for you and your computer from Sunshine. Just tick the box on the special order card.

Send to:

Sunshine Books,
12/13 Little Newport Street,
London WC2R 3LD.

Following the success of *Commodore 64 Machine Code Master,* Mark England and David Lawrence have set out once again to apply the power of machine code to the Commodore 64.

This time their aim has been to provide the reader with a host of practical machine code routines to release the sound and graphics capabilities of the Commodore 64.

The book contains a straightforward machine code loader to simplify the task of entering the machine code and each of the new routines is first simulated in BASIC and then fully explained alongside assembly language listings.

As an added bonus, as with their previous book, all the routines are added to the 64's BASIC as new commands, extending the capabilities of the machine far beyond the usual frontiers.

Mark England is a student of electronic engineering at the University of Southampton. He is author of the acclaimed Mastercode Assembler and co-author with David Lawrence of *Commodore 64 Machine Code Master.* David Lawrence is an established and popular author in the home computing field. His best-selling books for the 64 include *The Working Commodore 64* and *Advanced Programming Techniques for the Commodore 64.* He is a regular contributor to Popular Computing Weekly.



ISBN 0 946408 28 9

P 06 4
0522
151,7



This was brought to you

from the archives of

http://retro-commodore.eu