

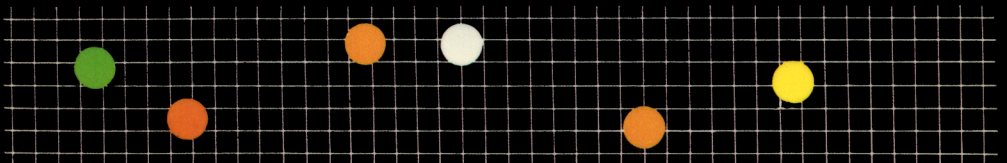
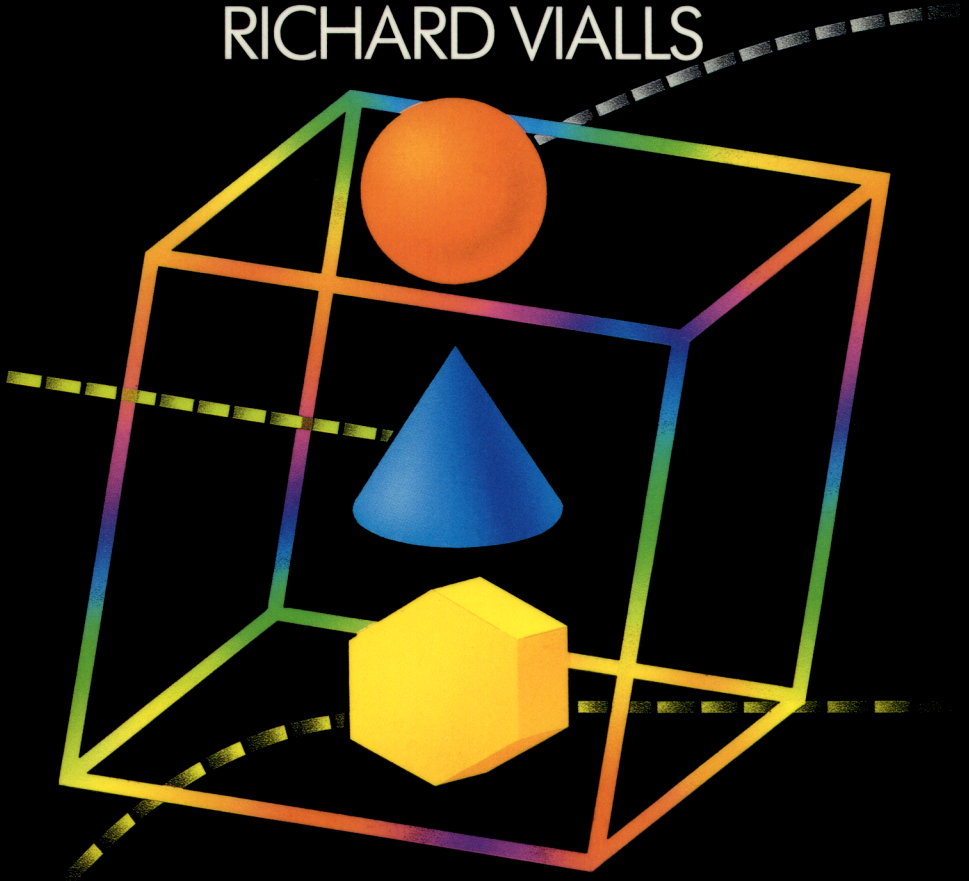
BBC MASTER GUIDES



MASTERING ASSEMBLY CODE

Advanced techniques for the BBC Model B Micro

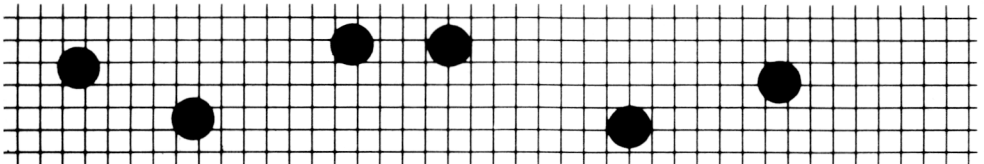
RICHARD VIALLS



BBC MASTER GUIDES

MASTERING ASSEMBLY CODE

RICHARD VIALLS



Throughout this book the letters 'BBC' refer to the British Broadcasting Corporation. The term 'BBC Micro' refers to the micro-computer manufactured by Acorn Computer plc under licence from the BBC. 'Tube' and 'Econet' are registered trade marks of Acorn Computer plc.

All rights reserved. No part of this book (except for brief passages quoted for critical purposes) may be reproduced or translated in any form or by any means, electronic, mechanical or otherwise, without the prior written consent of the copyright owner.

Disclaimer: Because neither the BBC nor the author have any control over the circumstances of use of the book, no warranty is given or implied as to its suitability for any particular application. No liability can be accepted for any consequential loss or damage, however caused, arising as a result of the information and advice given in this book.

© Richard Vials 1986
Edited by Meyer N. Solomon
Technical assistance by David Atherton
First published 1986

Published by the British Broadcasting Corporation
35 Marylebone High Street, London W1M 4AA

Typeset in 10/12pt. Rockwell Light by
August Filmsetting, Haydock.
Printed in England by R.J. Acford Ltd, Chichester.
Covers printed by The Malvern Press, London.
Bound by Dorstel Press Ltd, Harlow.

CONTENTS

	Preface	5
1	Assembly language programming	6
	Number systems	7
	The memory	13
	The CPU	15
	Commands	17
	Addressing modes	21
	Conditional branches	24
	The index registers	36
	Logical commands	39
	Indexed indirect addressing	42
2	The operating system	45
	Useful OS routines	45
	Memory usage	51
3	Pure machine code	55
	Addressing modes	56
	A machine code monitor	60
4	Interrupts	84
	The system VIA	87
	Events	97
	BRK	100
5	A few ways to protect your programs	103
	Locked tapes files	104
	Unlistable programs	105
	Disc tricks	108
6	The keyboard	113
	A BASIC input routine	114
	A machine code input routine	114
	The BREAK key	120

7	General graphics	123
	The graphics registers	123
	The video ULA	127
	Screen splitting	128
	Screen swapping	135
	A BASIC swap	139
	Three-dimensional graphics	143
8	Fill routines	149
	A BASIC fill	150
	A machine code fill	159
	A faster fill	167
9	Screen dumps	180
	A simple BASIC dump	180
	A machine code equivalent	184
	A colour-as-tone dump	188
	A miniature dump	199
10	Sprite graphics	216
	A BASIC sprite routine	217
	A machine code sprite routine	222
	Moving sprites	230
	The flicker licker	239
	Using the mover	252
	Anyone for tennis?	255
	Appendix A	276
	Two's-complement table	
	Appendix B	278
	Assembler commands and op-codes	
	Appendix C	279
	Op-codes and assembler commands	

PREFACE

This book is aimed at the programmer who has become proficient in BASIC and wants to explore the realms of machine code. The first section of the book sets out to give a detailed description of assembly language programming. However, it is impossible to *teach* someone to program creatively and professionally. The second section of the book discusses techniques and gives a series of examples of the uses of machine code. It is hoped that, by examining these programs in detail, you will begin to think in the ways that produce a good machine-code programmer.

Don't assume that the programs in this book are at a height of perfection. There are probably a number of improvements that can be made to them. Don't just use the programs in this book without thought. If, for example, you write an arcade game don't just use the sprite routine at the end of chapter ten – study that routine and then either write your own or adapt it to suit your particular needs. This will not only produce better programs, but should also help to make you a better programmer.

The overall message of this book is that a professional programmer is a perfectionist and will do everything within his power to improve a program to its limits.

ASSEMBLY LANGUAGE PROGRAMMING

Assembly code is not much more difficult to learn than BASIC. Machine code is the language the computer really understands underneath all the flashy BASIC commands. The heart of the BBC Micro is a Central Processing Unit (CPU) called the 6502. This CPU does all the 'thinking' and machine code is the language that this chip 'thinks' in.

BASIC is a 'high level' language. This means, simply, that it is far more sophisticated than machine code. For this reason, another chip is needed to interpret the BASIC commands, on a line-by-line basis, into machine code for the 6502 to understand. This chip is a Read Only Memory (ROM) with a machine code program permanently programmed into it. This program is called the BASIC *interpreter*. This must not be confused with a BASIC *compiler* which takes a BASIC program and converts this entirely into a machine code program so that the original BASIC can be scrapped and the faster machine code used instead.

Unfortunately, because BASIC has to be interpreted, it is very slow. If, however, we could talk to the 6502 directly, in its own terms, we could run programs much, much faster. The assembler is the means we use to talk to the 6502 directly. It effectively by-passes the BASIC interpreter. But before we can learn to use it, it is important to understand the terms *Binary* and *Hexadecimal*.

Number systems

In everyday life we use a number system that we call *decimal*. This is based on the idea that we count in tens. In decimal we have ten symbols that we use to represent the numbers zero to nine. To represent larger numbers we put these symbols together in a line with the furthest digit right representing the number of ones; the next one to its left representing the number of tens, then hundreds, and so on. So we can look at a number as if it were in a series of columns, each with a column heading saying what it represents.

1000s 100s 10s 1s
5 6 3 1

The computer, however, uses a system based on the idea of having only two symbols to count with—'0' and '1'. Again we use headings, but as the largest number the first column can represent is 1, the second column must count twos, the third fours, the fourth eights, etcetera. In other words, the column headings are powers of two. This counting system is called the binary system.

Binary	Decimal	Binary	Decimal
0	0	110	6
1	1	111	7
10	2	1000	8
11	3	1001	9
100	4	1010	10
101	5		

so 101011 in binary represents

1 times 1 = 1 (first column)
1 times 2 = 2 (second column)
0 times 4 = 0 (third column)
1 times 8 = 8 (fourth column)
0 times 16 = 0 (fifth column)
1 times 32 = 32 (sixth column)
—
43 in decimal

We call each digit in binary a *bit* (**binary digit**). The bit furthest left is called the *most significant bit* (MSB) because it has the largest column heading, and the bit furthest right is called the *least significant bit* (LSB) because it has the smallest column heading. This system is very long-winded but it suits the computer well, as the computer can represent 1 by a circuit being on, and 0 by a circuit being off.

Binary is not an ideal system for humans; for example, if we wanted to represent the decimal number 2141928901 in binary, it would be 111111101010110011110111000101—quite an eye-ful.

It is inconvenient to have the computer translate into decimal for us, so we need a counting system, half-way between binary and decimal, which is as easy for us to read as it is for the computer. Such a system is *hexadecimal*. To convert from binary to hexadecimal (or *Hex* for short), we split the binary number into groups of four bits (adding a few zeros on the left, if necessary, to make up complete groups of four bits). In each group there are sixteen possible combinations of 0s and 1s, so we assign each combination a symbol. Then, by running the symbols together, we have a means of representing the number.

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

So if we were to take our binary example of 111111101010110011110111000101 it would be coded like this:

0111	1111	1010	1011	0011	1101	1100	0101
7	F	A	B	3	D	C	5

This may well seem very complicated, but hexadecimal is like decimal and binary; only, instead of using base 10 or base 2, it uses base 16. In the decimal system, you 'carry' 1 into the next column to the left as soon as you reach ten in one column. In the binary system, you 'carry' 1 into the next column to the left as soon as you reach two in one column. In the hexadecimal system, you 'carry' one into the next column to the left as soon as you reach sixteen in one column. Imagine you have two, ten or sixteen fingers and you can't go wrong!

So that we don't get confused between decimal and hexadecimal, we put an & at the beginning of any hex number—thus decimal 2141928901 = &7FAB3DC5.

Just in case you aren't confused already, there is yet another system of counting used in computers – *binary coded decimal* (BCD for short). In this system, each digit of a decimal number is converted into a four-bit binary number. Then the four-bit sections are run together to make a BCD number, e.g.

2	1	4	1	9	2	8	9	0	1
0010	0001	0100	0001	1001	0010	1000	1001	0000	0001

Thus 2141928901 in decimal is

0010000101000001100100101000100100000001 in BCD. This number is slightly longer than its binary equivalent, but BCD is occasionally useful. For example, if you want to have a score in a game, you will want it to appear in decimal. By keeping the score stored in BCD it is then relatively easy to display it on the screen. If it is stored in binary it must first be converted to BCD before being displayed.

Now for a bit of terminology. The processor in the BBC Micro, the 6502, is an eight-bit processor. This means that it works in groups of eight bits at a time. With this system, the computer can only handle unsigned numbers from 00000000 to 11111111 (0 to 255 in decimal). This can be very annoying if you happen to want to use the number 256, but there are ways around this, as we will see later. Each group of eight bits is called a *byte*. Incidentally, each digit in

hex is called a nibble because it represents four bits. Four bits are half a byte—think about it!

Maths in binary is, in principle, exactly the same as maths in decimal.

Addition:

$$\begin{array}{r} 109 \quad 1101101 \\ + 38 \quad + 100110 \\ \hline 147 \quad 10010011 \end{array} \quad \text{remembering that } 1 + 1 = 10$$

Subtraction:

$$\begin{array}{r} 109 \quad 1101101 \\ - 38 \quad - 100110 \\ \hline 71 \quad 1000111 \end{array} \quad \text{remembering to borrow}$$

However, what happens when we get a negative number? To handle negative numbers, we use a system called *two's complement*. This method needs a fixed number of bits to each number. In the 6502, we use eight bits or one byte. To make a negative number, we take the positive value in binary and 'flip' each bit (0 becomes 1 and 1 becomes 0). This then has the disadvantage that -0 and $+0$ will have different codes, so we add 1 to our negative number to make -0 equal to 0.

Examples

-1 is represented by 00000001 flipped to 11111110 and then adding 1 = 11111111

-0 is represented by 00000000 flipped to 11111111 and then adding 1 = 00000000

In the second example, when adding 1 to 11111111, we should have got 100000000, but as the computer will only handle eight bits at a time, the ninth bit is lost, leaving 00000000. Let's try another example:

$$\begin{array}{r} -39 = 11011001 \quad (\text{in two's complement}) \\ 39 = 00100111 \end{array}$$

$$-39 + 39 =$$

$$\begin{array}{r} 11011001 \\ + 00100111 \\ \hline \end{array}$$

100000000 but we ignore the ninth bit,

so

$$-39 + 39 = 0$$

With this system, we don't use the most significant bit—the leftmost bit—as part of the number itself, but use it instead to show whether the number is positive (we set this bit to 0) or negative (we set this bit to 1). This bit is often called the sign bit. Thus the largest number we can store using the eight bits is 01111111, or 127, and the smallest number we can store is 10000000 or -128 (see Appendix A).

Of course, it is not very useful to have a computer which can only count from -128 to 127. However, if we use *two* bytes to store a number, and again use the most significant bit (the leftmost one) to show whether the number is negative or positive, we can count from 1000000000000000 (-32768) to 0111111111111111 (+32767).

However, the computer only provides for the adding of eight-bit numbers, and there could be a ninth-bit 'spill-over' when adding the most significant (leftmost) bits. So we need to use a thing called a *carry flag*. This is a single bit in the CPU which can either be set to 1 or cleared to 0. First we add the least significant bytes (or low bytes as they are sometimes called), and if we lose a ninth bit in the answer because of 'spill-over', the computer sets the carry flag to one to show this; otherwise, the computer clears it to zero. Then we add the two most significant bytes (or high bytes) and if the carry flag has been set to one from the addition of the low bytes then the computer adds one to the result. As we shall see later, the add command does most of this for us.

For example

High Bytes	Low Bytes	
01001001	01011010	(=18778)
+ 00100100	10110101	(= 9397)
<hr/>		
01101101	10000111	
+ 1		← Carry flag
<hr/>		
01101110		
<hr/>		
= 01101110	00000111	(=28175)

Thus the answer is 0110111000001111. Conveniently for us, when the computer adds two bytes together, it automatically adds in the carry flag and then puts the ninth bit of the answer, whether 0 or 1, back into the carry flag. This means that, if we want to, we can add three-byte numbers or even larger number to be taken away and flips all its bits. Then clear the carry flag before we add the low bytes. When we have just begun addition we have no carry to consider, so the carry flag must be cleared at the beginning of the addition.

Similarly, large numbers can be subtracted using the carry flag. However, with the subtract command, the carry flag is used as a *borrow* flag. Subtracting can be seen as adding the negative of a number; thus $27 - 5$ is the same as $27 + -5$.

The subtract command in machine code takes the number to be taken away and flips all its bits. Then it *adds* the two numbers together. Remember that, earlier, we said that we represented a negative number by flipping all its bits and adding one. If, *to begin with*, the carry is set to one, this provides the addition of one needed to complete the negative number in its two's-complement form. Thus the carry flag must be *set* to one to produce proper subtraction. As with the addition, the carry flag is set to the ninth bit of the result after the subtraction. Conveniently, this works out so that if a borrow does occur during this operation, the flag is cleared; otherwise it is set. This means that if we then add a set of high bytes the carry flag will ensure that any necessary borrowing is done.

Let's try, as a simple example, working out $27 - 5$.

$27 - 5$ in decimal = $00011011 - 00000101 = 22$

The computer takes the 00000101 and flips each bit to make the number 11111010. It then adds the 00011011:

```
    11111010
  +00011011
            
1  00010101
```

It then adds the carry *already* set to one.

$$\begin{array}{r}
 100010101 \\
 +1 \text{ from carry flag} \\
 \hline
 100010110
 \end{array}$$

The ninth bit is transferred to the carry flag leaving us with 00010110 which equals 22 in decimal.

Lets now try a two-byte example by subtracting 9397 from 18778.

The computer first takes the low byte of 9337 (10110101) and flips it to 01001010 then adds the low byte of 18778 (01011010). It also adds one from the carry flag (which we have previously set).

$$\begin{array}{r}
 01001010 \\
 +01011010 \\
 \hline
 10100100 \\
 +1 \text{ from carry} \\
 \hline
 010100101 \text{ this is low byte of answer.}
 \end{array}$$

The ninth bit of this answer (0) is transferred to the carry flag. The computer now takes the high byte of 9397 (00100100) and flips it to 11011011. It then adds the high byte of 18778 (01001001). It finally adds the carry flag which is now zero.

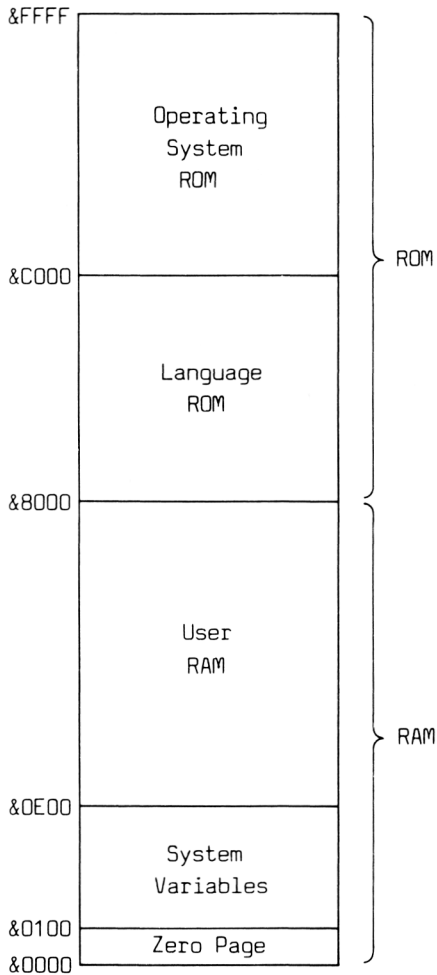
$$\begin{array}{r}
 11011011 \\
 +01001001 \\
 \hline
 100100100 \\
 +0 \text{ from carry flag} \\
 \hline
 100100100 \text{ this is high byte of answer.}
 \end{array}$$

The ninth bit is transferred to the carry flag. The answer is formed from the results of the two additions: 0010010010100101 or 9381 in decimal.

The memory

Let's now look at the memory. The computer does not know what any particular byte in its memory means. It is up to the programmer to decide, by

The Model B
memory map



giving the appropriate instructions, whether a particular byte represents a character of text, a binary number or a BCD number, the low or high byte of a two-byte number; or whatever.

Each byte of memory is numbered and given an address. An address is a number which, when given to the computer, tells the computer which byte of memory you are dealing with.

The 6502 uses two-byte numbers to represent addresses. The maximum number of different combinations that can be made with sixteen bits is two to the power of sixteen. Thus the 6502 can handle two

to the power of sixteen, or 65536 locations. These locations are each given an address number from 0 to 65535, and no two locations have the same address.

On the BBC Micro, the memory is split into two main sections. These are RAM (addresses 0 to 32767) and ROM (addresses 32768 to 65535). RAM is memory which we can alter and store our programs and data in; ROM is memory which cannot be changed.

The memory is divided up into *pages*. A page is all the memory locations that can be addressed with the same high byte. Thus locations &1200 to &12FF make up one page. These pages are numbered according to the high byte of the address, thus &100 to &1FF is page 1. The high byte is 1 and the low byte goes from &00 to &FF which means that a page consists of 256 bytes. Similarly, &0000 to &00FF is *zero page*. Zero page is used a lot for the storage of variables. This is because, as we shall see later, the machine code command for looking at zero page is shorter and faster than the normal commands.

The CPU

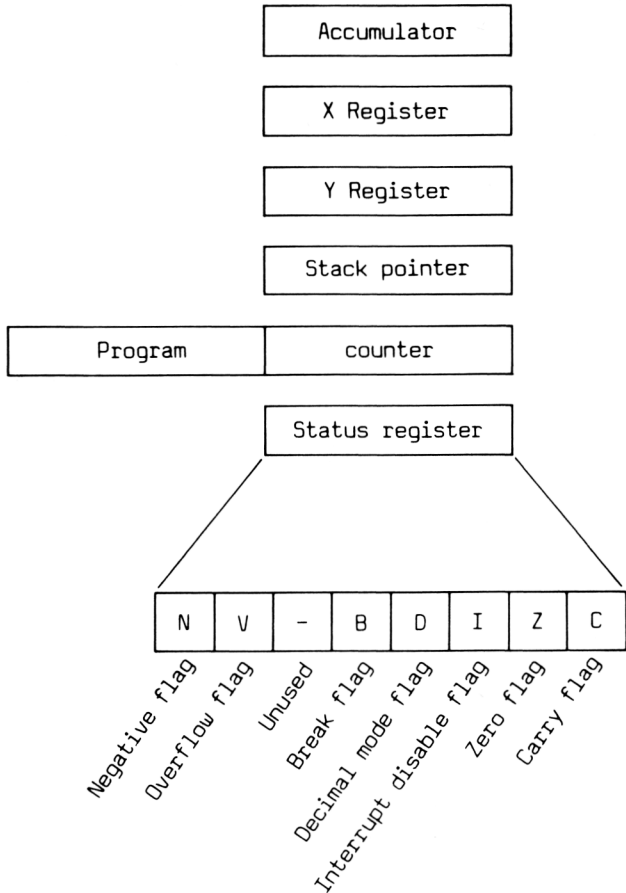
Before we can start to look at commands in machine code, we must look at the way in which the processor is organised.

As far as we are concerned the CPU contains six registers—very simple memories capable of storing one or two bytes. They are inside the processor, not part of the computer's main memory.

The most important one is the *accumulator*, or A register. This register is the one in which almost all the work is done.

Two other important registers, the X register and the Y register, are very useful for storing extra numbers we may need at any time—these are sometimes called the index registers.

Next is the stack pointer, which points into 256 bytes of memory, located from addresses &100 to &1FF, called the stack. This is a *first in last out* buffer: it behaves like a letter spike—the first thing you put on it is always the last to come off.



Then there is the program counter. In machine code each command is represented by either one, two or three bytes of memory. These bytes must be adjacent in the memory. The processor looks at the first byte, which is always the command byte and tells the computer what it is to do. The processor then knows (by examining the first, command, byte) how many bytes of data to expect after the command: some commands have no explicit data, some have one byte of data, others have two bytes. The next command must follow on straight after the data of the previous command otherwise the processor will not know where to look for it.

The program counter is a two-byte register

which holds the address of the memory location where the current command is stored. As the program runs, this register is automatically incremented by one every time a new byte of the program is loaded into the processor.

Finally, there is the status register. This register contains eight one-bit flags, one of which is not used. These flags can either be set (equal to 1) or clear (equal to 0) and are affected by some commands. Each shows that a particular circumstance has occurred. We have already met the carry flag which is affected by the 'add' and 'subtract' commands.

Commands

The accumulator is where all the work is done so we need a command to load the contents of a particular byte of memory into the accumulator. This command is called 'load accumulator'. It is stored in the memory as a one-byte number. However, to save us from having to remember the number that corresponds to each of the many machine code commands we use a program called an assembler to allow us to type something a little more understandable. It would be very cumbersome to have to type in a command like 'load accumulator' each time we want to use it, so each command in machine code is given a three-letter mnemonic which is easy to remember. The mnemonic for 'LoaD Accumulator' is LDA. So if we wanted to load the accumulator with the contents of memory address &1EFA, we would use the command:

```
LDA &1EFA
```

This command does not alter the contents of the memory, but makes a copy of the byte which is stored at &1EFA and places that in the accumulator. The previous contents of the accumulator are lost, having been replaced by the new byte.

Similarly, we need a command to store the contents of the accumulator in a specific memory location. This command is 'STore Accumulator'—STA. So, to store the contents of the accumulator at &3F5D, we would use:

This time the contents of the accumulator are not altered, but the previous contents of the memory location (in this case the contents of &3F5D) are lost, having been replaced by the byte from the accumulator.

So now we have two assembly code commands, but we need to know how to use them. They must be used as a program and we need to know how to use the assembler.

BBC BASIC contains a complete assembler which is very easy to use but you have to tell the BASIC interpreter where the assembler commands begin and end, and where in the memory to store the program. The easiest way to store the machine code program is to use the BASIC DIM command. If you use DIM without brackets, it reserves some memory which BASIC will keep totally free. This is ideal for putting machine code routines in. All you have to do is this:

```
10 DIM X% 100
```

Note that there are no brackets around the number.

This would reserve 100 bytes of memory. The address of the first of these bytes is put into the variable X%. It is sensible to use an integer variable for this job, as the address will always be an integer. However, the computer still does not know that this address is where we want the machine code to go. The computer will automatically start putting machine code at the location stored in the variable P%. So we ourselves have to set P% to the start of the reserved space, which is already stored in X%. Then we must tell the computer that from now on all commands will be in assembly code. The command to do this is [. So, before we can use the assembler, we must have something like this:

```
10 DIM X% 100 \ Reserve memory and store  
                starting address in X%  
20 P% = X%    \ Tell the assembler to store
```

```
30 [           the program at X%
                \ Following code is in assembly
                  language.
```

Now we can write our assembly language program. Once we have finished writing it, we must put a closing square bracket] to show that we are going back to BASIC. From now on, the shorter examples in this book will ignore the BASIC part of the assembler.

Look at the following program:

```
LDA &2000
CLC
ADC &2001
STA &2000
```

This program uses two commands we have not seen before—CLC and ADC. CLC is the mnemonic for 'CLear Carry' and, as its name suggests, it clears the carry flag. Remember that we have to do this before we can add any numbers together. ADC is 'ADd with Carry'. This command adds the contents of the memory location specified after it to the accumulator, using the carry flag as explained above. So this program takes the contents of address &2000, adds the contents of address &2001, and stores the result back at address &2000. However, the computer will not know what to do when it has done this, so we have to add one command, RTS, 'ReTurn from Subroutine', which tells the processor to go back to what it was doing before. So our small program would look like this:

```
10 DIM X%15
20 P%=X%
30 [
40 LDA &2000
50 CLC
60 ADC &2001
70 STA &2000
80 RTS
90 ]
```

We have dimensioned X% to reserve 15 bytes of memory. As the longest any command can be is 3 bytes, reserving three bytes for each command should leave ample room for this program. If you don't allow enough room, the program will very likely crash. Another point is that locations &2000 and &2001 are right in the middle of the program memory—if this routine were part of a long program, it would wipe out a part of itself. For most of your programming purposes, the BBC micro conveniently reserves 32 bytes of memory for storing variables. These 32 bytes are at locations &70 to &8F, so it is safest to use this memory where possible.

If we wanted the addition to be done in Binary Coded Decimal, we could preface it by the SED ('SEt Decimal mode') command which sets the decimal flag in the status register. When this flag has been set, all further addition and subtraction is done in BCD. However remember to use the CLD ('CLear Decimal mode') command afterwards to take the computer back into binary mode.

An assembly code program *does not* run as it is assembled. The assembler merely *encodes* the program into machine code and stores it away for future reference. To actually run the assembled code, use the CALL command, equivalent to GOSUB; only, you must specify the address of the start of the program rather than giving a line number. The RTS command at the end is the equivalent of the BASIC command RETURN. Here are some other commands:

DEC 'DECRe ment memory by one'—This subtracts one from the contents of the memory location specified.

INC 'INCRe ment memory by one'—This adds one to the contents of the memory location specified.

LDX 'LoaD X register from memory'—This copies the contents of a memory location into the X register.

LDY 'LoaD Y register from memory'—This copies the contents of a memory location into the Y register.

STX 'STore X in memory'—This copies the contents of the X register into a memory location.

STY 'STore Y in memory'—This copies the contents of the Y register into a memory location.

SBC 'Subtract memory from accumulator with carry'.

Remember that the carry flag must be set with SEC ('SEt Carry flag') before using SBC, and that, as with ADC, more than one byte may be subtracted.

Addressing modes

So far, we have seen that the command LDA &1EFA loads the accumulator with the contents of the memory location &1EFA. However, LDA can get a byte from the memory *in several different ways*.

Most machine code commands can be used in several different ways, called *addressing modes*, as they are the methods by which the processor finds the address of a byte to work on. The mode we have used up to now is called *absolute addressing*. However, there are thirteen different addressing modes which we can use, though not all can be used with each command. We have already seen one example of another addressing mode. CLC is an example of *implied* addressing. This mode is used in commands that do not need any explicit data to work upon. Other examples:

SEC 'SEt Carry'

RTS 'ReTurn from Subroutine'

INX 'INcrement X register'—This increases the contents of the X register by 1.

INY 'INcrement Y register'—This increases the contents of the Y register by 1.

DEX 'DEcrement X register'—This decreases the contents of the X register by 1.

DEY 'DEcrement Y register'—This decreases the contents of the Y register by 1.

(Note that if the X register contains &FF and the command INX is used, the answer should be &100; but, because the X register only has eight bits, the

largest number it can hold is &FF. Thus the ninth bit is lost (it is *not* transferred to the carry flag) so the result left in the X register is 0. Similarly, if the X register contains 0 and the DEX command is used, the result is 255. The same is true for INY and DEY.)

NOP 'No OPeration', just waste a tiny bit of time.

TAX 'Transfer Accumulator to X register'—The contents of the accumulator remain the same, X changes.

TAY 'Transfer Accumulator to Y register'—The contents of the accumulator remain the same, Y changes.

TXA 'Transfer X register to Accumulator'—The contents of the X register remain the same, A changes.

TYA 'Transfer Y register to Accumulator'—The contents of the Y register remain the same, A changes.

Notice that in implied addressing the mnemonic is not followed by any explicit data. The processor knows what the 'implied' data is.

Another useful addressing mode is *immediate* addressing. Here the byte of data actually used for the command to operate on is placed after the command with a 'hash' (#) mark to show that immediate addressing has been used. For example LDA #&CA would put the number &CA into the accumulator. If you examined the accumulator after using this command you would find &CA (decimal 202) in it.

It is often the case that you want to load the accumulator with the contents of a location whose address you don't actually know explicitly. Say, for example, you wanted to load the accumulator with the contents of the byte at PAGE. (PAGE is a variable that contains the address of the first byte of a BASIC program.) Normally this would be at &E00, but on disc machines it is at &1900. The BBC's assembler allows instructions such as LDA PAGE. This means that when the program is assembled, the computer will take the address to be whatever PAGE is currently set to. However, once assem-

bled, this cannot be changed. Similarly, complicated expressions can be used as addresses in the assembler, for example `LDX PAGE + (A% - 1)*2`.

Another command that is very useful is `JMP`—'JuMP to address'. This is the equivalent of the BASIC command `GOTO`, but it refers to an address in the memory rather than to a line number. This can be very inconvenient as we don't always know off-hand the address of the command we want to jump to. So the assembler provides another useful system, called *labels*. A label is a variable set to equal the address of a specific command. We precede the command by a full stop, then a variable name (something relevant, e.g. 'start', or 'sounds'), then a space. When the assembler comes across this, it sets the variable to equal the address at which the command is stored. This saves us the immense trouble of calculating the address ourselves. Then we can jump to the right address by simply using the variable name after the `JMP` command, e.g.

```
.start LDA &2000
      CLC
      ADC &2001
      STA &2000
      JMP start
```

However, though easy to read, it need not be set out like this. The assembler allows this sort of thing:

```
100.start LDA&2000:CLC:ADC&2001:STA&2000:JMPstart
```

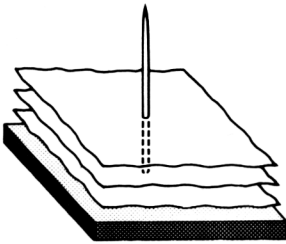
This saves memory and is easier to type in.

The equivalent of the BASIC command `GOSUB` is `JSR` ('Jump to SubRoutine') and is used like `JMP`. The BASIC command `CALL`, in fact, uses the `JSR` command to jump to a machine-code subroutine.

As the processor looks at a command it advances the program counter. Thus, by the time it has looked at a command to see what it has to do, the program counter will point to the beginning of the *next* command. When the processor comes across a `JSR` command, it takes the program counter (which points to the beginning of the next com-

mand, remember) and saves it on the stack as two bytes: low first, then high. In doing so, it moves the stack pointer to point to the next byte after the top of the stack. Then, at the end of the subroutine, when the processor meets an RTS command, it takes the top two bytes off the stack again and puts them back into the program counter. Thus the processor returns to the command *after* the JSR command. As the stack can hold 128 two-byte addresses, the *maximum* number of nested subroutines is 128. This decreases if the stack is also being used for other purposes.

It is interesting to note that the stack is stored in page one of the memory (&100 to &1FF) in such a way that &1FF is the first byte of the stack and all subsequent entries are stored in order backwards through the stack. It might help to think of an upside-down paper spike



It is also often useful to be able to save the contents of the accumulator on the stack and retrieve these contents again later; so two commands are provided to do this.

```
PHA 'PusH Accumulator onto stack'  
PLA 'PuLl Accumulator off stack'
```

There are also two useful commands for anyone wanting to 'edit' the stack, and these are:

```
TSX 'Transfer contents of Stack pointer to X  
register'  
TXS 'Transfer X register to Stack pointer'
```

(Notice that it is the stack pointer not the stack area that is transferred to the X register and vice versa.)

Conditional branches

It is useless to have a language without some form of the IF... THEN... conditional command. This is provided by making use of the flags.

There are four flags which are used for conditional jumps, or *conditional branches* as they are called in machine code. These are 'Carry', 'Zero', 'Negative' and 'Overflow'.

The zero flag, as its name suggests, is set when the result of some command is zero.

The negative flag is set if the result is negative—if bit seven is set (bit seven is the most significant bit of a byte); zero is treated as positive.

The overflow flag is more complicated. It is set either if there is a carry from bit 6 to bit 7 but the carry flag is not set; or if there is no carry from bit 6 to 7 and the carry flag is set. Strangely enough, there is a command CLV—'Clear oVerflow'—, but no 'set overflow' command. This flag is seldom used.

Once we have a flag set or cleared, we can do a conditional branch to another part of the program depending on the state (whether it is zero or one) of the flag. There are eight commands to do this:

BCC 'Branch if Carry Clear'

BCS 'Branch if Carry Set'

BEQ 'Branch if EQual to zero'

BNE 'Branch if Not Equal to zero'

BMI 'Branch if MInus'

BPL 'Branch if PPlus'

BVC 'Branch if oVerflow Clear'

BVS 'Branch if oVerflow Set'

These commands are followed by a one-byte positive or negative two's complement number. This gives the number of bytes to go forward in the program. If it is negative then the processor will go backwards. Thus BEQ &67 would move on &67 bytes if the result were zero, and BNE &FD (which could also be typed as BNE - 3) would go back three bytes if the result were not zero. It is difficult to calculate how many bytes to go forward to reach a particular command; so, again, labels may be used. However, because only one byte is used to contain the *offset*, the furthest back you can go is 128 bytes

(negatively); the furthest forward is 127 bytes (positively). Thus it is important to keep branches short. If necessary, you can do the following:

```
        BNE skip    \ if not carry on
        JMP equal   \ if so jump to equal
        .skip ....  \ carry on
```

Here, the routine *equal* is a routine we want to jump to if the zero flag is set. This routine is too far away from this section of the program to use a simple branch command. Instead we branch to *skip* if the zero flag is not set. Otherwise we jump to *equal* as JMP can reach anywhere in memory.

Conditional branches are very useful for delay routines. We have not yet used the X and Y registers much but they are often used for delays. If we want a very short delay, we can load the X register (or the Y register) with 0, then use the command DEX (which subtracts one from the X register and sets the negative and zero flags according to the result that is stored in the X register). This will leave 255 in the X register. The zero flag will therefore not be set. So if we do a BNE back to the DEX command (labelled as *loop* below), it will branch back. However, this time, the X register will contain 255. The result of all this is that the X register counts down from 0, then 255, all the way down to 0 again. When X finally reaches 0 again the zero flag is set and the BNE command fails to branch so the program carries on to the next command.

```
        LDX #0
        .loop DEX
        BNE loop
```

This produces a delay of about 0.0006 of a second. For a longer delay, we can create a nested loop around this using the Y register:

```
        LDY #0
        .Yloop LDX #0
```

```

•Xloop DEX
      BNE Xloop

      DEY
      BNE Yloop

```

This produces a delay of about 0.16 of a second. If we want an even longer delay, we have to press the A register into service as well. We don't have a 'decrement A' command so we have to use SEC and SBC #1:

```

      LDA #0

•Aloop LDY #0
•Yloop LDX #0
•Xloop DEX
      BNE Xloop
      DEY
      BNE Yloop

      SEC
      SBC #1
      BNE Aloop

```

This produces a delay of about 42 seconds which should be enough for most purposes. Different delays can be obtained by changing the values initially loaded into each register.

A problem occurs when labels are used, whether in jumps, subroutines or branches. As an example:

```

      BNE skip
      LDA #0
      STA &70
•skip DEX

```

The assembler obviously assembles the code in the order in which it is in the program. So, while assembling the above program, it will not yet have defined the variable *skip* when it gets to the command *BNE skip* because it will not have come across the label *skip* and so it gives the error 'No such variable'. To stop this happening, the BBC Micro's assembler can be used as a *two-pass assembler*.

In a two-pass assembler, the code is assembled twice. The first time, the assembler ignores any reference to labels it has not come across, at the same time leaving a space to fill in their values later. By the time it has finished the first pass it has found all the labels and so will have defined all the variables. Then, on the second pass, it assembles everything, including the reference to labels, because it now has all the addresses it needs stored in the variables. The assembler on the BBC Micro does not do this automatically, but it provides a useful command, OPT. This can only be used within the square brackets and is not assembled into machine code. It is thus known as a pseudo-operation. OPT is followed by a number or variable. Here is what the different numbers following OPT do:

- 0 Do not print assembled code and ignore errors.
- 1 Print assembled code and ignore errors.
- 2 Do not print assembled code and take note of errors.
- 3 Print assembled code and take note of errors.

If we set OPT to 0 or 1 on the first pass, and set it to 2 or 3 on the second pass, the errors that will naturally occur wherever there are 'forward' references to labels will be ignored on the first pass, while the labels are calculated; then any other errors will show up on the second pass. The easiest way to use OPT is by putting a FOR... NEXT... loop around the assembly code and setting OPT equal to the loop variable. When the assembler is enabled with the open brackets, OPT is automatically set to 3. Thus the first command within the brackets should be the OPT command. So the BASIC code to precede a section of assembly code now becomes:

```
10 DIM mc% 100
20 FOR pass% = 0 TO 3 STEP 3
30 P% = mc%
```

```

40 [OPTpass% \ On first pass, OPT is set
    .         \ to 0, so ignore all errors.
    .         \ On second pass, OPT is set
    .         \ to 3, so report any errors
assembly code \ and print assembled code.
    .
    .
    .
140 ]
150 NEXT
160 CALL mc%
```

OPT is not needed if all the branches, jumps and so on refer to *earlier* parts of the program—that is, if the program does not have any ‘forward’ references to labels. Note that P% must be reset to mc% at the beginning of each pass, because P% is incremented by the assembler as it assembles the code. This is sometimes useful as, after the] command, P% will point to the first free byte of memory after the machine code. For example, if you wish to save a piece of assembled code directly using *SAVE, P% will, after the code has been assembled, give the end address of the code.

A command that goes with the branch command:

CMP ‘CoMPare memory with accumulator’

This needs a little explanation. CMP subtracts the specified byte from the accumulator and sets the carry, zero and negative flags in the normal way. However, it doesn’t store the result in the accumulator as SBC does, so the accumulator is not altered. In fact the result is lost completely; all that is altered are the flags. The contents of the memory location are not altered either. Also, this command automatically sets the carry flag at the beginning, so you don’t have to worry about that.

The result of all this is that if the two numbers are equal, then taking one from the other leaves zero and the zero flag is set. If the accumulator is greater than the memory byte, then the carry will be set. The negative flag will be set equal to bit 7 of the result of the subtraction.

Probably the most common use of this command is for checking if the accumulator is equal to a particular value. You can put as many of these tests one after the other as you like because the accumulator is not affected! This could be very useful if, for instance, you are checking the GET command for the keys in a game—you could use a routine like this:

```
CMP # ASC("Z")
BEQ left
CMP # ASC("X")
BEQ right
```

There are also two equivalent commands for the index registers:

CPX 'Compare memory with X register'

CPY 'Compare memory with Y register'

Two other useful commands are:

PHP 'Push Processor status register'—Put contents of status register on stack.

PLP 'Pull Processor status register'—Pull byte from stack and place in status register.

These push and pull the current flag states to and from the stack and can be used to 'save' the results of a CMP command.

An addressing mode that is used a lot in the BBC Micro is *indirect addressing*. There is only one command that uses this mode, and that is JMP. Instead of jumping to the address specified after JMP, the CPU takes the byte contained at that address and the byte immediately following that and uses these two bytes as the low and high bytes respectively of the address the computer actually jumps to.

For example, there is a very useful subroutine stored in the Operating System ROM called OSWRCH (short for 'Operating System WRite CHaracter'). This is the routine BASIC uses for its VDU command (remember when using this that, as

with VDU, a carriage return does not also produce a line feed, i.e. VDU13 returns to the beginning of the same line). Thus if we wished to print a letter 'A' from the assembler, we would load the accumulator with the ASCII code for 'A' and then JSR to location &FFEE (where the routine starts). However, at this location there is an indirect jump instruction with the address &20E. The computer then jumps to the address pointed to by locations &20E and &20F (&20E contains is the low byte and &20F the high byte)—and this is where the actual routine starts. This system is sometimes called *vectoring*. In this example, locations &20E and &20F make up a vector. As &20E and &20F are in the RAM, we can change their contents so that, for example, the VDU command will jump to our own subroutine instead of the Operating System routine through our altering &20E and &20F to point to our own routine. This JMP command is typed with the address in brackets. e.g.

```
JMP (&20E)
```

One thing must be kept in mind when using this command. When the computer adds one to the address to fetch the high byte, it does not carry into the high byte of the address. Thus if we were to use the command JMP (&13FF), this would jump to the address stored at &13FF (low byte) and &1300 (high byte)! (&FF is incremented to &100 but the 1 isn't carried over to the high byte and so the result is &1300.) As an example of how to use the OSWRCH vector, the following program makes the computer always print a full stop instead of a space.

```
10 DIM X% 100
20 FOR A%=0 TO 3 STEP 3
30   P% = X%
40   [
50   OPTA%
60   .start CMP #ASC(" ") \ Check if char is
                        space
70           BNE print    \ if not, print it
80           LDA #ASC(".") \ if so, load full
                        stop
```

```

90 .print JMP (&230) \ then go to main
                           routine
100 ]
110 NEXT
120 ?&230=?&20E           :REM Copy vector into
130 ?&231=?&20F           :REM &230 and &231
140 ?&20E= start MOD256   :REM then set to
150 ?&20F= start DIV256   :REM new routine.

```

The \ symbol is equivalent to the BASIC REM statement—the computer ignores everything else on that line.

Lines 120 and 130 make a copy of the VDU vector in a spare vector (&230) that the operating system does not use, so that we can jump to the original OSWRCH routine to print a character.

This program can be very useful for checking for spaces accidentally typed at the end of lines, but remember to press BREAK to clear it before attempting to edit lines, otherwise the spaces you really want will become full stops.

Although there are no proper multiplication and division commands in machine code, there are two commands for multiplication and division by 2:

ASL 'Arithmetic Shift Left memory'

LSR 'Logical Shift Right memory'

Their names don't give much of a clue as to how they work. ASL takes all the bits in a byte and shifts them one place to the left. The least significant bit (the rightmost one) becomes zero and the most significant bit is placed in the carry flag. The original contents of the carry flag are thus lost.

For example,

```

C  7 6 5 4 3 2 1 0
0  1 1 0 1 0 1 1 1

```

becomes, after ASL

```

C  7 6 5 4 3 2 1 0
1  1 0 1 0 1 1 1 0

```

The result of this is to multiply the byte by two.

LSR does the opposite. It shifts all the bits to the right, sets the most significant bit to zero and shifts the least significant bit into the carry. Again, the original contents of the carry are lost, e.g.

```
7 6 5 4 3 2 1 0 C
1 1 0 1 0 1 1 1 0
```

becomes, after LSR

```
7 6 5 4 3 2 1 0 C
0 1 1 0 1 0 1 1 1
```

The result of this is to divide the byte by two.

Apart from absolute addressing, these two commands can be used with another addressing mode called *accumulator* addressing. As its name implies, in this mode the byte used and affected is the one contained in the accumulator and not in a memory location. This mode is used from the assembler by placing the letter *A* after the command.

```
ASL A
LSR A (The spaces are not necessary)
```

The remaining two commands that can use accumulator addressing mode are:

```
ROL 'ROtate Left memory'
ROR 'ROtate Right memory'
```

These commands are similar to ASL and LSR but they do not lose the original carry contents entirely. ROL shifts the byte left. As it does so the old contents of the carry are shifted into bit zero and the old contents of bit 7 are shifted into the carry. Thus the whole byte plus the carry flag—nine bits in total—is rotated one bit to the left, e.g.

```
C 7 6 5 4 3 2 1 0
1 0 0 1 0 1 1 1 0
```

becomes, after ROL

C 7 6 5 4 3 2 1 0

0 0 1 0 1 1 1 0 1

ROR is the exact opposite, e.g.

7 6 5 4 3 2 1 0 C

0 0 1 0 1 1 1 0 1

becomes, after ROR

7 6 5 4 3 2 1 0 C

1 0 0 1 0 1 1 1 0

By combining ASL and ROL commands, a number of two or more bytes can be multiplied by two.

1 The least significant byte of the number is multiplied using ASL. This leaves its most significant bit in the carry.

2 The next byte up is multiplied with the ROL command. The carry resulting from step 1 is thus now placed in bit zero of the second byte and bit seven of the second byte is left in the carry, ready for another ROL command on the next byte up (if any).

For example,

High byte

Low byte

00110101

10101101 = 13741 (decimal)

V

V

ROL

ASL

V

V

C

C

0 < 01101011 < 1 < 01011010 = 27482 (decimal)

The original number is doubled.

Similarly, numbers comprising two or more bytes can be divided by two by using LSR on the high byte first then ROR on successive lower bytes, e.g.

High byte Low byte

01101011 01011010 = 27482 (decimal)

V

V

LSR

ROR

V

V

C

C

00110101 > 1 > 10101101 > 0 = 13741 (decimal)

The original number is halved.

If the original number is odd, this method will lose the 'half' at the end of the answer into the carry flag, i.e. the carry flag contains the remainder after division by two. This can be used to test if a number is even or odd by testing the carry flag.

As an example of what can be done with these four commands, let's write a short program that will multiply the contents of the accumulator by 10 and store the answer at &70 (low) and &71 (high).

To do this we must first multiply by five and then multiply by two. To multiply by five we can multiply the number by four and then add the original number to make five lots of the original number.

The first thing to do is this:

```
    .mten    STA &70
            LDX #0
            STX &71
```

The original number is now stored as a two-byte number in &70 (low) and &71 (high). We will need to add the original number to the answer later, so we can leave the original number in the accumulator. Next, we can multiply the contents of &70 and &71 by 4 by multiplying them by 2 twice.

```
    ASL &70
    ROL &71
    ASL &70
    ROL &71
```

We now have four times the original number in &70 and &71. To make five times the original number we must add the accumulator (which *still* contains the original number) to &70 and &71.

```
    CLC
    ADC &70
    STA &70
    BCC skip
    INC &71
    .skip    ...
```

Note that because we are adding zero to the high byte, it is quicker to use a branch and an INC command than to use a further ADC command.

We now have five times the original number in &70 and &71. Finally, we need to multiply this by two to get 10 times the original number.

```
.skip    ASL &70
         ROL &71
         RTS
```

The index registers

The index (X and Y) registers are used a lot for what is known as *absolute indexed* addressing. This mode is similar to absolute addressing, but an X or Y is placed after the address (separated from it by a comma). The contents of the register are added to the address to form the actual address from which a byte is fetched or saved. This is very useful for arrays where the index register can be used to point into a one-dimensional array up to 256 elements long. Here is an example of this:

```
10 DIM mc% 30
20 FOR pass% = 0 TO 3 STEP 3
30 P% = mc%
40 [OPT pass%
50 .start LDX #0
60 .loop LDA array ,X
70      JSR &FFE3
80      INX
90      CPX #8
100     BNE loop
110     RTS
120 .array EQU$ CHR$13+"Hello."+CHR$13
130 ]
140 NEXT
150 CALL start
```

The command EQU\$ is one of four similar commands available only with BASIC II which are for storing data in the middle of assembly code.

The first is EQU\$. This must be followed by a one-byte number. This byte is then inserted into the

middle of the assembly code. Make sure that the processor will never try to run this byte as an instruction—if it did, it would almost certainly crash. This byte can then be used as a byte of *data*.

For example, if you have run out of room in the variable space from &70 to &8F, you could put a series of EQUW commands after the main program each preceded by a label and then use the labels as free memory locations.

The second command is EQUW which is identical except that it uses a two-byte number. This number is stored low-byte first.

The third is EQUW which uses a four-byte number. This is stored low-byte first.

The fourth is EQUW which uses a string. This is stored in the order the characters appear in the string.

In the program above, the string 'Hello.' between two carriage returns is placed at the label *array* and can then be referred to by the program, a byte at a time, using the X register to point into the string.

For those of you with BASIC I these four commands are not available. To get around this you will need to replace them with these pieces of code:

```
100 .temp    EQUW &CA
```

is replaced by:

```
100 .temp ]
102 ?P%=&CA
104 P%=P%+1
106 [OPTpass%
```

Because P% always points to the beginning of the next machine code command we can exit the assembler and place the correct byte in the memory in the correct place. We then need to increment P% to point to the next byte. Finally we can re-enter the assembler (remembering to set OPT as this is always reset to 3 on entry to the assembler).

Likewise for the other three commands:

```
100 .temp EQUW &1CA3
```

is replaced by:

```
100 .temp ]
102 ?P%=&A3:P%?1=&1C
104 P%=P%+2
106 [OPTpass%
```

and

```
100 .temp EQUW &12345678
```

is replaced by:

```
100 .temp ]
102 !P%=&12345678
104 P%=P%+4
106 [OPTpass%
```

```
100 .temp EQUW "HELLO"
```

is replaced by:

```
100 .temp ]
102 $P%="HELLO"
104 P%=P%+LEN"HELLO"
106 [OPTpass%
```

The subroutine at &FFE3 is a ready-made operating system routine. It checks whether the accumulator contains a carriage return and, if so, prints both a carriage return and a line feed; if not, it jumps to OSWRCH which prints the character in the accumulator. This routine is called OSASCI.

Another example of a simple use of the index registers is for filling a block of memory locations. The next example clears a Mode 7 screen. By changing the character the program uses, the screen may be filled with any character.

```
10 MODE 7
20 DIM mc% 30
30 FOR A% = 0 TO 3 STEP 3
```

```

40 P% = mc%
50 [OPTA%
60 .clear LDA #ASC(" ")
70         LDX #0
80 .loop  STA &7C00,X
90         STA &7D00,X
100        STA &7E00,X
110        STA &7F00,X
120        DEX
130        BNE loop
140        RTS
150 ]
160 NEXT
170 CALL clear

```

In this program, because the screen takes up 1K (which is 4 pages) starting at &7C00, one absolute indexed X command can only clear one quarter of the screen, as the X register can only go from 0 to 255; so four commands are used, one for each page. Of course, this whole program could be replaced by a call to OSWRCH with the accumulator containing 12. (This is the ASCII code for 'Clear Screen'.)

Logical commands

A very useful command is:

ORA 'OR Accumulator with memory'

This works on each bit of the two bytes (one in the accumulator, the other from the memory) separately. If either of the corresponding bits in the two bytes is 1 or both are 1, then the corresponding bit in the answer is 1. If, however, they are both 0, then the answer will be 0. The result goes back into the corresponding bit of the accumulator. This OR function can be used to set particular bits in a byte to 1 and leave the others untouched.

first bit	second bit	answer bit
0	0	0
0	1	1
1	0	1
1	1	1

E.g.

01010110

OR

00001111

01011111

A similar command is

AND 'AND accumulator with memory'

This command only sets the bit in the answer to 1 if the corresponding bits in the first byte AND the second byte are *both* 1. The AND function can be used to set particular bits in a byte to 0 leaving the others untouched. Both this and the OR command are sometimes called MASK commands because they mask particular bits to 1 or 0.

first bit	second bit	answer bit
0	0	0
0	1	0
1	0	0
1	1	1

E.g.

01010110

AND

11110000

01010000

The third and last command along these lines is the EOR or Exclusive-OR command. This sets the answer bit to 1 if one of the corresponding bits in the first byte OR the second byte is one, but not if they are both one or both 0. This command can be used to flip particular bits in a byte from 1 to 0 or 0 to 1 leaving the others untouched. This is how the processor flips all the bits to make a negative number when doing a subtract command.

first bit	second bit	answer bit
0	0	0
0	1	1
1	0	1
1	1	0

E.g.

01010110

EOR

00001111

01011001

01010110

EOR

11111111

10101001

Notice that, as using EOR with 255 flips all the bits, a second use of EOR with 255 will flip them all back again leaving the original number. This is very useful in games graphics. A figure in a game can be Exclusive-OR'ed with the screen to put it on and then Exclusive-OR'ed again to remove it.

BIT does the same as AND but doesn't place the answer back in the accumulator. It forgets the answer, but it does set the status register flags. If the answer is zero then the zero flag is set (otherwise it is cleared). Also, bits 6 and 7 (the most-significant-but-one and the most significant) of the byte taken from the memory are placed in the overflow and negative flags respectively. This can be used in a variety of ways. By using BIT on a byte in the memory, bits 6 and 7 can be tested using the flags. But its main use is that, by setting just one particular bit of one of the numbers to 1, leaving the others at zero, and using BIT on the two numbers, the zero flag will then indicate whether the corresponding bit in the other number is 0 or 1.

E.g.

00010000 (in accumulator)

BIT

01010101 (in memory)

00010000 which is not zero, so the zero
flag will be clear

00010000 (in accumulator)

BIT

10100101 (in memory)

00000000 which is zero, so the zero
flag will be set

Indexed indirect addressing

The four remaining machine code commands are concerned with *interrupts*; they are covered in chapter four. There are still, however, two addressing modes to be covered.

The first of these is *post-indexed indirect addressing*. In this mode, a one-byte number is specified after the command, and this refers to an address in zero page. The processor takes the byte at this address and the byte at the address after it to form a new address. As in indirect addressing, the low byte is stored first followed by the high byte. Also, if the address specified is &FF, the two bytes used will be the ones at &FF (low byte) and &00 (high byte). Once the processor has this address, it adds the contents of the Y register (this mode can only be used with the Y register) to it. This now forms the address of the byte the processor actually works on. Within the assembler, the zero page address is put in brackets with a comma and a Y after the brackets, e.g.

LDA (&70),Y

If location &70 contains, say, &00 and location &71

contains &30, the address 'pointed to' would be &3000. If the command is used with Y ranging from 0 to 5, then locations &3000, &3001, &3002, &3003, &3004 and &3005 would be successively addressed.

This mode uses just two zero page locations to hold a variable that itself points into the memory. For example, if we wished to clear a Mode 0 screen, we place the address of the beginning of the screen in, say, &70 and &71. Then we could use a loop of Y to store zeros at 256 locations starting with that address, then add 256 to the address and repeat the procedure until the screen is fully cleared, e.g.

```
.clear LDA #0      \ Place start of
        STA &70    \ screen address
        LDA &&30    \ in &70 (low)
        STA &71    \ and &71 (high)
        LDA #0     \ Fill with 0
        LDY #0     \ start Y at 0
.loop   STA (&70),Y \ store on screen
        INY        \ next byte
        BNE loop   \ 'til 256 done
        INC &71    \ then next 256
        BIT &71    \ check bit 7 of &71
        BPL loop   \ if clear repeat
        RTS        \ if set, then &8000
                        reached therefore
                        screen
                        cleared so end.
```

The last addressing mode is *pre-indexed indirect addressing*. It uses the X register only. The X register is added to the zero page address itself rather than to the address stored at that zero page address. In other words, the processor takes the one-byte address after the command, adds the X register (it ignores any carry so that the result is still a zero page address). Then it takes the byte at that address and the byte at the address after it to form a two-byte address (as before, low then high) and this is the address of the byte that it uses. This command is typed in the assembler like this:

STA (&70,X)

If X is 5, this instruction would form the actual address from the contents of &75 (low) and &76 (high).

Note that the X is *within* the brackets this time. The use of this is that a table of addresses can be formed in zero page and perhaps contain the addresses of a series of missiles on the screen in a game. However, because only limited zero page locations are usually available to you as a machine code programmer, the applications for this mode on the BBC Micro are very limited.

There are some commands that people assume can be used but which the computer does not allow. For instance, there is a command ORA for the accumulator, but there are no equivalent commands for the X and Y registers. Another point is that not all commands support all addressing modes. For instance, the command INC A is not valid—to do this you will have to use the commands:

CLC

ADC #1

THE OPERATING SYSTEM

Assembly code is very difficult to use on its own because it contains no specific input or output commands. This means that, say, printing a character on the screen requires a series of LDA and STA commands to place the relevant bytes in the relevant places in the memory. This would prove difficult even in Mode 7 let alone in Mode 2! If, further, we want to use one of the more complicated pieces of the hardware on the machine such as the disc drive then the machine code needed will become ridiculously complicated. The *operating system* ROM in the computer comprises just under 16K of machine code routines which will handle virtually all the input and output operations you are ever likely to need. Not only does this ROM contain all the *FX and other 'star' commands, but it also contains routines for handling VDU commands and filing commands, to name just a few.

Useful OS routines

Some of the operating system routines are rarely used or are too complicated to cover completely in this book. See *The Advanced User Guide* (Cambridge Microcomputer Centre, 1983). If you are in doubt, the rule is that *any* input or output command that is available from BASIC can be accessed through one of the operating system routines somehow. Here, however, are some of the more useful routines.

OSBYTE

(Operating System BYTE operator routine)
Location: &FFF4

This routine is used to set up any of a large number of flags that the operating system uses to decide what to do in particular situations. It is the equivalent to the BASIC *FX command. There are over 150 of these commands though most of them are not particularly useful. There is a list of most of the useful commands on pages 418–441 of the *User Guide*. However, for a complete list of all the OSBYTE calls, and what they do, see *The Advanced User Guide*.

To use the OSBYTE routine from machine code the accumulator must be set to the number of the specific command you wish to use and the X and Y registers must be set to any parameters that the routine needs for that particular command. OSBYTE is at address &FFF4.

OSWORD (Operating System WORD operator routine)
Location: &FFF1

This routine is similar to OSBYTE but it handles operations that require larger amounts of data. This data is stored in a CONTROL BLOCK. This is a series of bytes which contain the parameters needed for a particular OSWORD call. To use OSWORD you should set aside a block of memory (a parameter block) long enough for the OSWORD command you want to use, and then place the parameters in this block. Then set the accumulator to the number of the particular OSWORD command you want to use and set the X and Y register to the low and high bytes respectively of the address of the first byte of the parameter block. Then call the routine, which starts at &FFF1.

For a complete list of all the OSWORD calls, and what they do, see *The Advanced User Guide*.

OSWRCH (Operating System WRite CHaracter).
Location: &FFEE

This is the routine that performs the equivalent of a BASIC VDU command. By loading the accumulator with a number and calling &FFEE the contents of the accumulator will be written to the screen. *All* output

to the screen can be directed through this routine. For example, to clear the text screen the following code should be used:

```
LDA #12
JSR &FFEE
```

To take another example, the BASIC PLOT command is accessed using the sequence VDU25, plot number, low byte of X coordinate, high byte of X coordinate, low byte of Y coordinate, high byte of Y coordinate. Thus to enter graphics Mode 4 and draw a line from the bottom left-hand corner of the screen to the top right-hand corner of the screen, the following piece of code could be used:

```
LDA #22 \ Change mode
JSR &FFEE \ to
LDA #4 \ mode 4
JSR &FFEE \
LDA #25 \ PLOT
JSR &FFEE \
LDA #4 \ 4,
JSR &FFEE \
LDA #0 \
JSR &FFEE \ 0,
LDA #0 \
JSR &FFEE \
LDA #0 \
JSR &FFEE \ 0
LDA #0 \
JSR &FFEE \
LDA #25 \ PLOT
JSR &FFEE \
LDA #5 \ 5,
JSR &FFEE \
LDA #&FC \
JSR &FFEE \ 1276,
LDA #4 \
JSR &FFEE \
LDA #&FC \
JSR &FFEE \ 1020
LDA #3 \
JSR &FFEE \
```

However, this would be somewhat tedious. A point to note is that the OSWRCH routine exits with the A, X and Y registers unchanged; which means that we can do this:

```
LDX #0
.loop LDA table,X
      JSR &FFEE
      INX
      CPX #14
      BNE loop
      RTS
```

We can then place, starting at the address pointed to by *table*, a table containing the following bytes:

```
22,4,25,4,0,0,0,0,25,5,&FC,4,&FC,3
```

Make sure you understand this technique as it is very useful!

For a complete list of what the VDU codes do, see pages 377–389 of the *User Guide*.

One important thing to note about this routine is that, as with the VDU command, a carriage return (ASCII code 13) returns the cursor to the beginning of the line it is already on—it *does not* move the cursor down to the beginning of the next line. To do this you must send a carriage return followed by a line feed (ASCII code 10). This can be a nuisance, so the operating system provides another two routines.

The first of these is OSNEWL (Operating System NEW Line) which is at address &FFE7. This routine sends a carriage return and a line feed to OSWRCH.

The second is OSASCI (Operating System ASCII output routine) which is at address &FFE3. This routine is the same as OSWRCH except that, if a carriage return is sent to it, it does both a carriage return and a line feed. It is interesting to see how this is actually done by the operating system. Here is the relevant section:

```
.OSASCI CMP #13
        BNE OSWRCH
```

```
•OSNEWL LDA #10
          JSR OSWRCH
          LDA #13
•OSWRCH  JMP (&20E)
```

Notice that OSWRCH jumps via the vector at &20E to the routine that actually prints a character.

OSRDCH (Operating System Read Character)—the GET routine.

Location: &FFEO

This is the routine that the BASIC interpreter uses to get values for the BASIC GET command. After calling the OSRDCH routine the carry flag will be set if an error has occurred; the accumulator will then contain the error number. This will usually only occur if the ESCAPE key has been pressed, in which case the accumulator will contain 27. Here, the program must acknowledge this by calling OSBYTE with the accumulator set to &7E (see page 429 of the *User Guide* or page 149 of *The Advanced User Guide*).

If the carry flag is clear then the accumulator will contain a character that has been read from the current input device. This will usually be the keyboard although, in some cases, it could be the RS423 interface; or a disc file if a *EXEC command has been used, for example.

If no key has been pressed the routine will wait until a key has been pressed before it returns a value, so it cannot be used for 'arcade' games. OSBYTE with the accumulator set to &81 should be used (see page 153 of the *Advanced User Guide*). So, if we want to get a key from the keyboard, the following piece of code should be used:

```
JSR OSRDCH \ Get a key
BCC next   \ if no error process key
CMP #27    \ if it isn't escape
BNE error  \ goto error routine
LDA #&7E   \ if escape
```

```

        JSR OSBYTE \ acknowledge and
        JMP escape \ goto escape routine
.next   ...      \ process key

```

If we are using the keyboard then the error can only be an escape, so we can use this:

```

        JSR OSRDCH \ Get a key
        BCC next   \ if no error process key
        LDA #&7E   \ if escape
        JSR OSBYTE \ acknowledge and
        JMP escape \ goto escape routine
.next   ...      \ process key

```

If the escape key has been disabled then all that is needed is a call to OSRDCH.

OSCLI (Operating System Command Line Interpreter)

Location: &FFF7

This is the routine the operating system uses to process * commands. If in BASIC you use a command preceded by a * then the BASIC interpreter uses this routine.

To use this routine you need to have the command you want to perform stored in memory as an ASCII string. At the end of the string there should be a carriage return (ASCII code 13). Then you should set the X and Y registers to the low and high bytes respectively of the address of the first character of the string in the memory. Then you should call the OSCLI routine, which starts at &FFF7.

As an example, let us take a game program which loads into the computer and then loads a Mode 2 graphics screen before running the game. If we assume that the screen has been saved after the main program on the tape or disc using *SAVE SCREEN 3000 8000 then the following code can be used:

```

        LDA #22           \ Mode command:
        JSR &FFEE         \
        LDA #2            \ Change to
        JSR &FFEE         \ Mode 2.

```

```

LDX #str MOD256    \ set X and Y to
LDY #str DIV256    \ start of string.
JSR &FFF7          \ call OSCLI.
...                \ rest of game.
...
.str EQU$ "LOAD SCREEN" \ string with carriage
EQU$ 13            \ return at end.

```

Note that a * is not needed at the start of the string. This routine can be used for all * commands. However, as we have already seen, there is an easier way to perform *FX commands.

Memory usage

Normally on the BBC Micro the user is allowed to use the memory stretching from PAGE to HIMEM. This can be used for BASIC programs, machine code programs, variables, data, etcetera. However, it is sometimes necessary to place a piece of machine code somewhere where the BASIC cannot affect it. For example, you might have a machine code routine that was used by several different BASIC programs that chain each other. This routine must be kept clear of the BASIC or it will become corrupted. There are several methods of doing this. The most obvious method is to set HIMEM lower, leaving room for the machine code to be placed just above it. However, if the screen mode is changed, this resets HIMEM according to the new mode and may clear the machine code. It is usually better instead to set PAGE higher to leave room for the machine code just below it, as PAGE is only reset on BREAK. However, this means you need a loader program that sets the value of PAGE, and some programs may reset PAGE for other reasons anyway.

However, there is a large amount of memory set aside for the operating system and the BASIC. On a tape machine PAGE is normally set to &E00 leaving 3.5K for the ROMs to use. Now, since not all this memory is likely to be in use at any one time, it is usually possible to use some of it to place machine

code routines in. This 3.5K is described below, a page (256 bytes) at a time.

- Zero page This page should be used for machine code variables only, unless you are very short of memory. Locations &00-&8F are reserved for the current language. However, BASIC itself does not use locations &50-&8F of this so these locations are safe to use. Locations &90-&9F are allocated to the ECONET system, so, unless you are using ECONET, these are safe. Locations &A0-&A7 are used by the NMI interrupt which is used by the disc system. However, on tape machines this is not used and is safe. On disc machines this MUST NOT be used. The rest of zero page is used by the operating system and should not normally be used.
- Page 1 This is the processor's hardware stack. However, the processor will not normally use more than the top quarter of the stack so it is reasonably safe to use &100-&1BF though I would recommend that you only use &100-&17F and then only for *temporary* storage of strings, etcetera.
- Page 2 This is the operating system's main work area and as such should not be used.
- Page 3 This contains some more operating system workspace. Memory locations &300-&37F contain the VDU command workspace; &380-&3DF are used by the cassette system and &3E0-&3FF make up the keyboard input buffer. None of this is particularly safe to use.
- Page 4 This is used by BASIC for variable storage. Locations &400-&46B contain the values of the integer variables @%-Z%. These are stored in order, using four bytes for each. Each of these is stored low byte to high byte, as a four-byte two's complement number. The integer variables can be useful for passing variables between BASIC and machine code. The rest of this page (&46C to &4FF) is used for pointers which indicate where the other variables are stored.

If BASIC is going to be used, this page cannot be used for machine code. If, however, you are going to write a program—such as a game—which will not use BASIC, and you are short of memory, this page can be used.

Page 5 This is used by BASIC as a stack for FOR, REPEAT and GOSUB return addresses. Again, this can only be used if BASIC is not needed.

Page 6 This is used by BASIC for working on strings. So long as BASIC is not working on strings at a particular time, this could be used as a temporary work space. However, this page must be clear before returning to BASIC from your machine code routine. As before, if BASIC is not needed, this page is safe to use for machine code.

Page 7 This is the BASIC line input buffer. Again it can be used safely if BASIC is not used.

Page 8 This is laid out as follows:

&800–&83F Sound workspace

&840–&87F Sound buffers

&880–&8BF Printer Buffer

&8C0–&8FF Envelope storage

If any of these sections are not in use then they are safe to use for machine code.

Page 9 This is used in three *different* ways.

1)

&900–&9BF Extra sound envelopes

&9C0–&9FF Speech buffer

2)

&900–&9BF RS423 output buffer

&9C0–&9FF Speech buffer

3)

&900–&9FF Cassette output buffer.

If none of these is in use then this page can be used.

-
- Page 10 This is either the cassette or the RS423 input buffer. As before, this page can be used if the cassette and RS423 systems are inactive.
- Page 11 This is used for soft key definitions. If you use this to store your own code, the soft keys will produce rubbish if pressed. This can be countered by disabling the soft keys using *FX225.
- Page 12 This is used for the user-defined characters. This page can be used so long as the user-defined characters are not printed.
- Page 13 Memory block &D00-&D9E is used by the NMI system. Cassette users may use this but disc users must use *TAPE first. Memory block &D9F-&DEF is the expanded vector set. This is used by some paged ROMs and by the disc system to vector useful calls. With care, tape users can use it. &DF0-&DFE is used by the ROMs for workspace allocation and should not be used except with extreme caution.
- All this means that, if you are writing a program that uses none of the system's buffers and does not use BASIC, you have available over 2K more than usual for machine code commands. With great care, even short BASIC programs can be placed in pages 8-12 by setting PAGE accordingly.

PURE MACHINE CODE

We have seen in the last two chapters how to program in assembly code. However, this language is completely artificial; it is not one that the computer understands directly. We have to convert an assembly code program into machine code, using an assembler, before we can use it. For most purposes this is ideal for us as it means we don't have to understand pure machine code. If, however, you are looking at someone else's program you may not have the original assembly code but only the machine code. In some circumstances it is easier to write in pure machine code—if you can do it! This chapter will show you how assembly code is converted into pure machine code.

A machine code program is stored in the memory as a series of consecutive bytes. Each instruction takes up either one, two or three bytes depending on what addressing mode it uses.

The first byte of an instruction tells the CPU which command is being used and also which addressing mode it is being used with.

There is a specific one-byte code for each available combination of command and addressing mode. This byte is called the *op-code*. Because not all the addressing modes can be used with each command the total number of legal op-codes is 151. If you use a code that is not legal the computer will usually ignore it, though some illegal codes produce strange results. This is because there are some commands on the 6502 which are not documented because either they don't work properly or are totally useless.

If your computer has a 65C02 processor, which is

a recent improvement on the old 6502 processor, then it will have 59 extra legal op-codes. The standard BBC Micro, however, has only the 151 standard 6502 op-codes. There is a complete list of the legal op-codes and what they do in Appendices B and C.

Addressing modes

Here is a list of all the addressing modes we can use and what data (if any) is needed for each.

Implied addressing

In this mode no explicit data is needed, so the command only uses the one byte for the op-code.

Immediate addressing

In this mode one byte of data is needed. Thus the assembler command `LDA #&7E` becomes:

```
A9 7E
```

Note that the data is stored in the memory as the byte directly after the command byte.

Accumulator addressing

In this mode there is no explicit data involved. The `A` (for accumulator) after the command in assembly code is in fact implied by the op-code itself. The `A` is only for our convenience. Thus `LSR A` becomes:

```
4A
```

Absolute addressing

In this mode two bytes are needed after the op-code to specify the address of the byte the processor must work on. These address bytes are always stored with the low byte first, followed by the high byte. Thus `STA &FE62` becomes:

```
BD 62 FE
```

Zero page addressing

If we wanted, for example, to store the accumulator at zero page address `&78` using absolute addressing we would need:

```
BD 78 00
```

However, there is an addressing mode for just this sort of situation. By using *zero page addressing* the processor knows we are using zero page and we only have to send one byte of data to the processor. Thus the example above becomes:

85 78

This means that if we store all our frequently used variables in zero page we can access them slightly faster and with a saving of one byte of program per command. The assembler does this automatically—if it is faced with an absolute addressing command with an address in zero page, it automatically uses zero page addressing.

Absolute X addressing

In this mode the contents of the X register are added to the address before it is used. In terms of machine code the command is identical to absolute addressing but the op-code is different. Thus STA &6435,X becomes:

9D 35 64

Absolute Y addressing

This uses the Y register but is otherwise identical to absolute X addressing except that the op-code is different. For example LDA &900,Y becomes:

89 00 09

Zero page X
addressing

As with absolute addressing there is a zero page version of absolute X addressing. This is zero page X addressing and uses only one byte of data. This command automatically works in zero page. Thus LDA &78,X becomes:

85 78

Zero page Y
addressing

This is the same as zero page X addressing, only with a different op-code. For example STX &90,Y becomes:

96 90

Indirect addressing	This mode (which can only be used with the JMP instruction) has two bytes after it which together form the address in which the processor looks for the actual address. Thus JMP (&230) becomes:
	6C 30 02
Pre-indexed indirect addressing	In this mode a zero page address is specified so it only uses one byte of data. Thus ADC (&70,X) becomes:
	61 70
Post-indexed indirect addressing	This again only needs a one-byte zero page address. Thus STA (&84),Y becomes:
	91 84
Relative addressing	This mode is the most complicated of all. It needs one byte of data in the form of a positive or negative number. This is the number that is added to the program counter if the condition being tested is true. The program counter, of course, determines which command is being carried out or executed. The way to calculate this <i>offset</i> , as it is called, is to subtract the address of the first byte of the command you want to branch to, from the address of the first byte of the next command after the branch command.

For example, take the assembly code program:

```

                LDX #&80
    .loop      DEX
                BNE loop
                RTS

```

If we assembled it at address &2000 onwards and then looked at it we would find the following:

Address byte

```

2000  A2  \ LDX #
2001  80  \ &80

```

```
2002 CA \ DEX
2003 D0 \ BNE relative
2004 FD \ &2002-&2005=-3=&FD
2005 60 \ RTS
```

(a branch backwards)

```
INC &70
BNE nocarry
INC &71
.nocarry RTS
```

becomes:

```
Address byte
2000 E6 \ INC
2001 70 \ &70
2002 D0 \ BNE
2003 02 \ &2006-&2004=2
2004 E6 \ INC
2005 71 \ &71
2006 60 \ RTS
```

(a branch forwards).

Notice that the assembler OPT command has an option so that you can see an assembled listing of the code as it is assembled. This can be enabled by using OPT3 for the second pass. Below is the second of our example programs and the print-out it produces.

```
10 HIMEM=&2000
20 FOR pass%=0T03 STEP3
30 P%=&2000
40 [OPT pass%
50     INC &70
60     BNE nocarry
70     INC &71
80 .nocarry RTS
90 ]
100 NEXT
```

```
2000     OPT pass%
```

```
2000 E6 70   INC &70
2002 D0 02   BNE nocarry
2004 E6 71   INC &71
2006 60      .nocarry RTS
```

A machine code monitor

Now that we have seen how pure machine code works, we need a way of using it. On the BBC Micro there is no quick way to look at a section of memory. If we want to look at a machine code program or write one, we need a 'window' into the memory. The program we need is called a machine code monitor. It is relatively easy to write a simple monitor in about 600 bytes, so that is exactly what we will do.

In this and the following chapters we will discuss each program in detail. Each section of the program will be, in general, followed by numbers in brackets that refer to line numbers in the full listing that follows the description.

This program will work in Mode 7. We need to be able to look at a whole section of the memory, say 200 bytes, and not just one byte at a time. The best way to do this is to display the contents of the memory in hexadecimal as a table on the screen. So as to get a large number of bytes on the screen we will need to have eight bytes on each line of this table. We need to be able to see, at a glance, the address of each byte, though we can make do with only displaying the address of the first byte on each line. So we will end up with a display looking like:

```
78A0 01 20 E3 FF 20 E3 FF A9
78A8 00 A2 07 20 E3 FF CA D0
78B0 FA A9 04 A0 C0 4C F4 FF
78B8 C9 30 90 33 C9 3A B0 06
78C0 38 E9 30 4C D1 78 C9 41
78C8 90 25 C9 47 B0 21 38 E9
78D0 37 85 75 A4 74 B1 70 0A
78D8 0A 0A 0A 05 75 91 70 B1
78E0 70 20 C2 79 A9 08 20 E3
78E8 FF 20 E3 FF 4C 61 78 C9
78F0 9F F0 19 C9 9C D0 63 20
78F8 23 7A C6 74 A5 74 C9 FF
7900 F0 03 4C 61 78 A9 07 85
```

```
7908 74 4C 0F 79 20 23 7A A9
7910 1F 20 E3 FF A9 00 20 E3
7918 FF 20 E3 FF A9 0B 20 E3
7920 FF A5 70 38 E9 08 85 70
7928 A5 71 E9 00 85 71 A5 70
7930 38 E9 60 85 72 A5 71 E9
7938 00 85 73 20 DE 79 A9 1F
7940 20 E3 FF A9 00 20 E3 FF
7948 A9 18 20 E3 FF A9 20 A2
7950 27 20 E3 FF CA D0 FA 4C
7958 61 78 C9 9E F0 1D C9 32
```

We can also make the program more pleasant to use by making the addresses (at the extreme left) yellow and the data (the contents of the memory) green. We can then highlight the byte we are currently working on by making it white.

We are going to use the cursor keys and the shifted cursor keys for control of the cursor, so the first thing we need to do is set up the cursor keys to do this.

We could produce ASCII codes from the cursor keys using *FX4,1 but this will not distinguish between normal and shifted cursor keys. The way round this is to set up the cursor keys as soft keys and set them to generate ASCII codes. We do this with two commands. First, *FX4,2 sets up the cursor keys as soft keys 12 to 15. The shifted cursor keys will now automatically produce ASCII codes (from &8C to &8F) but the normal cursor keys will not. So, second, we make the normal cursor keys generate ASCII codes by setting the ASCII base of the normal function keys to &90 with *FX225,144. This means that instead of producing strings the soft keys will generate ASCII codes of &90 plus the key number, so that the normal cursor keys will now generate codes &9C to &9F. So the start of our machine code routine looks like this:

```
.monitor LDA #4
        LDX #2
        LDY #0
```

```
JSR osbyte
LDA #&E1
LDX #&90
JSR osbyte
```

(80-140)

Notice that we have used *osbyte* instead of &FFF4. To do this we must define the variable *osbyte* at the beginning of the assembler program. We are also going to use the OSASCI and OSRDCH routines so we can define these at the same time. We also need a place to put the machine code monitor program. For our purposes let's put it at &7900 and move HIMEM down to leave room for it. This will cause problems if you are writing a graphics program but it is easy, in that event, to change the program to assemble the machine code elsewhere in the memory.

So the beginning of the assembler routine looks like this:

```
10 HIMEM=&7900
20 osasci=&FFE3
30 osbyte=&FFF4
40 osrdch=&FFE0
50 FORpass%=0T02STEP2
60 P%=&7900
70 [OPTpass%
```

Now back to the machine code. Our next task is to go into Mode 7 and turn the cursor off.

```
LDA #22
JSR osasci
LDA #7
JSR osasci
LDA #23
JSR osasci
LDA #1
JSR osasci
LDA #0
LDX #8
.loop1 JSR osasci
```

DEX
BNE loop1

(150-270)

Notice that the simple BASIC command VDU23,1,0;0;0;0; becomes quite complicated in machine code. The eight zeros are easier to send with a loop unlike in BASIC.

Next we must decide what memory address we are interested in displaying. It is convenient to set this to &0000 for the moment as it is easy to step through the memory to the location we are interested in. Because of the way we are going to display the table, we are going to be treating the memory as an array eight bytes across by 8192 bytes down.

It will be easier if we keep the address of the *first* byte on the line we are looking at separate from the number of the byte (0-7) within that line.

Let's say that &70 and &71 contain the address of the first byte on the line where the cursor is at present positioned and &74 contains the number of the byte on that line which we are interested in (0-7). Initially the address of the first byte of the line and the number of the byte on the line will both be zero, so we can add to our program:

```
LDA #0  
STA &70  
STA &71  
STA &74
```

(280-310)

Next we must display a screenful of memory—24 lines of eight bytes each. It would be sensible to set up a routine which just prints one line (eight bytes) of memory, and use this repeatedly. However, before we can even do this we need a routine that will display the value of one byte as two hex digits. For this routine let us specify that the byte to be printed must initially be found in the accumulator. We will have to work on one nibble (half a byte) at a time, so we have to save the complete byte while we

work on the first nibble in the accumulator. We can save the accumulator in &75. Next we can mask out the least significant nibble, leaving the most significant nibble in the accumulator (this will be the left-hand digit of the hex byte). We will then have to shift it right four times so that we get a number from zero to fifteen in the low nibble of the accumulator.

```
.byte    STA &75
         AND #&F0
         LSR A
         LSR A
         LSR A
         LSR A
```

(1990-2040)

Next we need to display this digit on the screen. We will need to do this twice (left-hand nibble and right-hand nibble) for each byte so we need a separate routine called *nibble* to do this.

Having called this routine we must reload the accumulator with the original byte and this time mask out the most significant nibble, leaving the right-hand nibble in the least significant nibble of the accumulator, and call the *nibble* routine again. However, note there is little point in calling *nibble* again as, once it is called, the *byte* routine will have finished so the next command would be RTS. We may as well 'fall through' straight to *nibble* and let the RTS at its end (supplied by the OS subroutine OSASCII) do the job. This means that we have to place *nibble* directly in place of the second *JSR nibble* command. This leaves us with:

```
         JSR nibble
         LDA &75
         AND #&F
.nibble  ...
```

(2050-2080)

The *nibble* routine must add 48 (ASCII code for 0) to the number in the accumulator before printing it

using OSASCI. However, if the number is 10 or more (decimal) then we need first to add a further seven to bring it to the corresponding ASCII codes for the characters A, B, C, D, E and F.

```
.nibble  CLC
          ADC #48
          CMP #58
          BCC print
          CLC
          ADC #7
.print   JSR osasci
          RTS
```

Notice that because the last-but-one command of *nibble* is a JSR we can instead just jump to OSASCI and the RTS command at its end will save us from needing an extra RTS at the end of *nibble*. Thus the end of *nibble* becomes:

```
.print   JMP osasci

(2080-2140)
```

We now have the byte display routine; so, next, we need to write the line display routine. For this we need the address of the first byte on the line. This may not necessarily be the line the cursor is on, so we can't use &70 and &71. Instead we can specify that the address of the first byte on the line must be stored at &72 and &73. The routine must first of all print a 'yellow' teletext code for the address. Then it must print the two-byte address stored at &72 and &73 (remember that &73 is the high byte).

```
.line    LDA #&83
          JSR osasci
          LDA &73
          JSR byte
          LDA &72
          JSR byte
```

```
(2160-2210)
```

Next we need a space to separate the address from the data.

```
LDA #32
JSR osasci
```

(2220-2230)

We will next use *post-indexed indirect addressing* to load the byte to be displayed into the accumulator; so we need to set Y to zero for the first byte. Then, for each byte, we can print a 'green' teletext code to separate the byte from the previous one; then load the byte into the accumulator and display it; then increment Y; and repeat the process until all eight bytes that make up the line have been displayed. After that we only need a carriage return to complete this routine. As before, we can save ourselves from putting an RTS at the end by jumping to the OSASCI routine.

```
LDY #0
.loop3 LDA #&82
      JSR osasci
      LDA (&72),Y
      JSR byte
      INY
      CPY #8
      BNE loop3
      LDA #13
      JMP osasci
```

(2240-2330)

Having written a routine for displaying a line, we can now go back to the main routine and display a whole screenful of data. For this overall display, it would be best if the byte we are currently examining or altering always appears half-way down the screen as then we can see what we have done and what is coming. For this reason the line the cursor is on will always be the thirteenth line down. Thus to print the block of memory above the cursor we need to subtract 96 (12 times 8) from the contents of

&70 and &71. This we can put in &72 and &73 ready to display a line. We also need to start at the top of the screen (later we will jump back to this point so the change of mode is not sufficient).

```
.display LDA #30
          JSR osasci
          LDA &70
          SEC
          SBC #96
          STA &72
          LDA &71
          SBC #0
          STA &73
```

(320-400)

We are going to print 24 lines in one go, so we can use the X register to count down from 24 to 1. We also need to add 8 to the contents of &72 and &73 to move the address forward by eight bytes after each line.

```
.loop2   LDX #24
          JSR line
          LDA &72
          CLC
          ADC #8
          STA &72
          LDA &73
          ADC #0
          STA &73
          DEX
          BNE loop2
```

(410-510)

We are now at the stage where we need a cursor to appear. We are going to highlight the byte we are interested in by making it white. To do this we need to put a 'white' teletext code before it and a 'green' teletext code after it. As we are going to want to remove this cursor again, it would be sensible to use a subroutine to position the text cursor where

we are going to place the 'white' byte. We shall call this routine *cursor1*. We know that the cursor will always be on line 12 so we only have to calculate how far across it will be. As each byte uses up three screen characters for its display we need to multiply the byte number by three. To do this we need to load the accumulator with the contents of &74, shift the accumulator left one bit to multiply it by two, and then add the contents of &74 to the accumulator to make three times the original number. We then have to add six to the accumulator to shift the cursor right past the address at the beginning of the line. We can use `VDU31,x,y` to move the text cursor on the screen. So the routine looks like this:

```
.cursor1 LDA #31
          JSR osasci
          LDA &74
          ASL A
          CLC
          ADC &74
          CLC
          ADC #6
          JSR osasci
          LDA #12
          JMP osasci
```

(2350-2450)

We can now go back to the main routine. Firstly we need to call *cursor1* and then we need to print a 'white'. As we have used 'green' codes to separate the bytes we don't need to put another one in. The text cursor is then on the first nibble of the byte.

```
.start   JSR cursor1
          LDA #&B7
          JSR osasci
```

(520-540)

We now need to get a key from the keyboard using OSRDCH. If ESCAPE has been pressed we must acknowledge it with OSBYTE &7E and we then want

to turn the cursor back on with VDU23,1,1;0;0;0;
reset the cursor keys to their normal functions with
*FX4,0 and move the cursor to the bottom of the
screen using VDU31, all before returning to BASIC.

```
.key      JSR osrdch
          BCC key1
          LDA #&7E
          JSR osbyte
          LDA #23
          JSR osasci
          LDA #1
          JSR osasci
          JSR osasci
          LDA #0
          LDX #7
.loop4    JSR osasci
          DEX
          BNE loop4   \ Note that X must now
          LDA #4      \ be zero, so we don't
          LDY #0      \ need to set it for
          JSR osbyte  \ the OSBYTE call.
          LDA #31
          JSR osasci
          LDA #0
          JSR osasci
          LDA #24
          JMP osasci
```

(550-770)

Having checked for the ESCAPE key we need to see if the byte under the cursor is being altered. If the key pressed is either 0 to 9 or A to F then we must alter the byte accordingly. Firstly we can check if the code is less than 48. If so, then we must check for other keys.

```
.key1     CMP #48
          BCC key2
```

(780-790)

Next we can look to see if the code is less than 58. If

so, then a number key has been pressed and we need to subtract 48 to get the value of the new nibble.

```
CMP #58
BCS letter
SEC
SBC #48
JMP hex
```

(800-840)

If not, we then want to look for a letter. If the code is less than 65 we are not interested and must wait for another key. If it is larger than or equal to 71 then we must look to see if it is a cursor key. Otherwise we want to subtract 55 to get the value of the new nibble.

```
.letter CMP #65
        BCC key
        CMP #71
        BCS key2
        SEC
        SBC #55
```

(850-900)

We now have to decide what to do with the nibble. Probably the best way to input the byte is for each new nibble to shift the old byte left one nibble. Thus the high nibble is lost, the low nibble becomes the high nibble and the nibble typed in replaces the low nibble. To do this we have to temporarily store the nibble we have typed in while we shift the memory byte left four bits. Then we can OR in the new nibble and store the result back in the memory.

```
.hex   STA &75
        LDY &74
        LDA (&70),Y
        ASL A
        ASL A
        ASL A
        ASL A
```

```
ORA &75
STA (&70),Y
```

(910-990)

This program will allow us to look into the ROMs but if we try to store an alteration back into a ROM nothing will happen. To make sure that the user doesn't think something has happened it would be sensible to load the byte back again before storing it on the screen. This way, if the byte hasn't been altered then the displayed byte on the screen won't change. As we have left the text cursor at the first nibble on the screen we can just call the subroutine *byte* to display the byte, move the text cursor back again and go back to waiting for the next key.

```
LDA (&70),Y
JSR byte
LDA #8
JSR osasci
JSR osasci
JMP key
```

(1000-1050)

If the key pressed is not a number or a letter we must check whether it is a cursor key and act accordingly. We shall check first of all for a cursor-up. If this has been pressed we shall branch to the relevant routine. Otherwise we shall check for a cursor-left.

```
•key2  CMP #&9F
        BEQ up
        CMP #&9C
        BNE key3
```

(1060-1090)

Having established that the cursor-left key has been pressed we need to remove the highlight cursor. We will have to do this several times so we need a subroutine which we shall call *cursor*. This must

first call *cursorl* to position the text cursor and then rub over the highlight teletext code with a 'green' code.

```
    .cursor JSR cursor1
           LDA #&82
           JMP osasci
```

(2470-2490)

So our *cursor-left* routine can now remove the cursor. Next it must decrement &74 to move the address of the byte being looked at, back by one. If this is now 255 then we must set it to seven (the end of the line) and do a *cursor-up*. Otherwise we can go back to the start of the main routine. Notice that as this last branch is more than 128 bytes we have to use the 'skip and jump' technique.

```
           JSR cursor
           DEC &74
           LDA &74
           CMP #255
           BEQ skip1
           JMP start
    .skip1 LDA #7
           STA &74
           JMP up1
```

(1100-1180)

While we are at it, we can write the *up* routine as well. This will be the same as *up1* but with a call to *cursor* in front of it.

```
    .up     JSR cursor
    .up1    ...
```

(1190)

Next we have to scroll the screen down one line. We can do this by first moving the cursor to the top of the screen using *VDU30* and doing a *cursor-up*.

```
.up1    LDA #30
        JSR osasci
        LDA #11
        JSR osasci
```

(1200-1230)

Next we have to subtract eight from &70 and &71 to move the cursor line back by one.

```
LDA &70
SEC
SBC #8
STA &70
LDA &71
SBC #0
STA &71
```

(1240-1300)

Next we must call *line* with the address of the top line in &72 and &73. This will be the contents of &70 and &71 minus 96.

```
LDA &70
SEC
SBC #96
STA &72
LDA &71
SBC #0
STA &73
JSR line
```

(1310-1380)

Lastly we need to clear the bottom line of the screen by moving to the bottom line and printing 31 spaces.

```
LDA #31
JSR osasci
LDA #0
JSR osasci
LDA #24
```

```
        JSR osasci
        LDA #32
        LDX #31
    .loop5 JSR osasci
           DEX
           BNE loop5
           JMP start
```

(1390-1500)

Next we must check for cursor-down. If this has been pressed then we must jump to the relevant routine.

```
    .key3  CMP #&9E
           BEQ down
```

(1510-1520)

Otherwise we must check for cursor-right. If you are typing a long program it will be annoying to have to find the cursor-right key between typing in each byte, so we shall allow both the cursor-right key and the space-bar to do the same job.

```
        CMP #&9D
        BEQ right
        CMP #32
        BNE key4
```

(1530-1560)

The *right* routine must first remove the old cursor and then increment &74 to move the cursor right by one byte. If the contents of &74 now equal eight then we must set it back to zero and jump to the cursor-down routine. Otherwise, we must use the 'skip and jump' technique to branch back to *start*.

```
    .right JSR cursor
           INC &74
           LDA &74
           CMP #8
           BEQ skip2
```

```
                JMP start
.skip2         LDA #0
                STA &74
                JMP down1
```

(1570-1650)

As with the *up* routine, we can insert the *down* routine here.

```
.down         JSR cursor
.down1        ...
```

(1660)

Next we have to scroll the screen up a line. We can do this by moving to the bottom of the screen and doing a cursor-down. However, at the same time we can print the new bottom line. This is because we are leaving a blank line at the bottom of the screen. By printing the new line in this blank space the carriage return at the end will scroll the screen for us. First we need to move to this blank line.

```
.down1        LDA #31
                JSR osasci
                LDA #0
                JSR osasci
                LDA #24
                JSR osasci
```

(1670-1720)

Now we add eight to the address of the cursor line.

```
LDA &70
CLC
ADC #8
STA &70
LDA &71
ADC #0
STA &71
```

(1730-1790)

Next we must store the address of the new bottom line in `&72` and `&73` and call `line`. Then we can jump back to `start`.

```
LDA &70
CLC
ADC #88
STA &72
LDA &71
ADC #0
STA &73
JSR line
JMP start
```

(1800-1880)

We now have all we need. However, if you want to look at the contents of address `&7F00` you will have to scroll through from `&0000` to `&7F00` one line at a time and this will be slightly tedious. To solve this problem we shall make shifted cursor-up and shifted cursor-down move a page at a time through the memory. This we can do by incrementing or decrementing the high byte of the address of the cursor line (`&71`) and jumping back to `display`.

```
.key4    CMP #&8E
          BNE key5
          INC &71
          JMP display
.key5    CMP #&8F
          BNE key6
          DEC &71
          JMP display
```

(1890-1960)

Finally, if the key that has been pressed is not one we are interested in then we must go back and wait for another key to be pressed.

```
.key6    JMP key
```

(1970)

This program is a very simple one. There are several ROMs available that have more sophisticated monitors in them. BBC *Monitor* (ROM based) from BBC Publications (1985) is one such. Another package from the same publishers is *Toolbox 2* by Ian Trackman (1985), which comprises a book and software tape. It has a particularly interesting implementation of a monitor among its many utilities, and complements this book.

Of course, you can improve our monitor program. However, it will provide a useful tool for those people who can't be bothered to write a better version or buy a ROM. It will also give you valuable experience in how assembly code programs work.

Here, then, is a complete listing of the monitor program. You might like to use the command *SAVE MONITOR 7900 +20B 7900 to save the machine code once it is assembled. Then the monitor can be used by typing *MONITOR. This will take up less room on a disc or tape and will be faster to load. However, you should keep a copy of the source assembly code program so that you can assemble the program into different places, if necessary, by changing P% at line 60.

You do not need to type in the comments on the right-hand side of the listing.

```
10 HIMEM=&7900
20 osasci=&FFE3
30 osbyte=&FFF4
40 osrdch=&FFE0
50 FORpass%=0T02STEP2
60 P%=&7900
70 [OPTpass%
80 .monitor LDA #4          \ Main program
90           LDX #2          \ initialisation.
100          LDY #0
110          JSR osbyte
120          LDA #&E1
130          LDX #&90
140          JSR osbyte
150          LDA #22
160          JSR osasci
```

```

170          LDA #7
180          JSR osasci
190          LDA #23
200          JSR osasci
210          LDA #1
220          JSR osasci
230          LDA #0
240          LDX #8
250 .loop1   JSR osasci
260          DEX
270          BNE loop1
280          LDA #0
290          STA &70
300          STA &71
310          STA &74
320 .display LDA #30      \ Display sect of
330          JSR osasci   \ memory as table 8
340          LDA &70      \ bytes by 24
350          SEC          \ lines.
360          SBC #96
370          STA &72
380          LDA &71
390          SBC #0
400          STA &73
410          LDX #24
420 .loop2   JSR line
430          LDA &72
440          CLC
450          ADC #8
460          STA &72
470          LDA &73
480          ADC #0
490          STA &73
500          DEX
510          BNE loop2
520 .start   JSR cursor1 \ Start checking
530          LDA #&87     \ keys.
540          JSR osasci
550 .key     JSR osrdch
560          BCC key1     \ Check for
570          LDA #&7E     \ ESCAPE.
580          JSR osbyte
590          LDA #23
600          JSR osasci

```

```

610          LDA #1
620          JSR osasci
630          JSR osasci
640          LDA #0
650          LDX #7
660 .loop4   JSR osasci
670          DEX
680          BNE loop4
690          LDA #4
700          LDY #0
710          JSR osbyte
720          LDA #31
730          JSR osasci
740          LDA #0
750          JSR osasci
760          LDA #24
770          JMP osasci
780 .key1    CMP #48      \ Check for byte
790          BCC key2      \ being altered.
800          CMP #58
810          BCS letter
820          SEC
830          SBC #48
840          JMP hex
850 .letter  CMP #65
860          BCC key
870          CMP #71
880          BCS key2
890          SEC
900          SBC #55
910 .hex     STA &75
920          LDY &74
930          LDA (&70),Y
940          ASL A
950          ASL A
960          ASL A
970          ASL A
980          ORA &75
990          STA (&70),Y
1000         LDA (&70),Y
1010         JSR byte
1020         LDA #8
1030         JSR osasci
1040         JSR osasci

```

1050		JMP key	
1060	.key2	CMP #&9F	\ Check for
1070		BEQ up	\ cursor-up.
1080		CMP #&9C	\ Check for
1090		BNE key3	\ cursor-left.
1100		JSR cursor	
1110		DEC &74	
1120		LDA &74	
1130		CMP #255	
1140		BEQ skip1	
1150		JMP start	
1160	.skip1	LDA #7	
1170		STA &74	
1180		JMP up1	
1190	.up	JSR cursor	
1200	.up1	LDA #30	
1210		JSR osasci	
1220		LDA #11	
1230		JSR osasci	
1240		LDA &70	
1250		SEC	
1260		SBC #8	
1270		STA &70	
1280		LDA &71	
1290		SBC #0	
1300		STA &71	
1310		LDA &70	
1320		SEC	
1330		SBC #96	
1340		STA &72	
1350		LDA &71	
1360		SBC #0	
1370		STA &73	
1380		JSR line	
1390		LDA #31	
1400		JSR osasci	
1410		LDA #0	
1420		JSR osasci	
1430		LDA #24	
1440		JSR osasci	
1450		LDA #32	
1460		LDX #31	
1470	.loop5	JSR osasci	

```

1480          DEX
1490          BNE loop5
1500          JMP start
1510  .key3    CMP  #&9E    \ Check for
1520          BEQ down     \ cursor-down.
1530          CMP  #&9D    \ Check for
1540          BEQ right    \ cursor-right
1550          CMP  #32
1560          BNE key4
1570  .right   JSR cursor
1580          INC  &74
1590          LDA  &74
1600          CMP  #8
1610          BEQ skip2
1620          JMP start
1630  .skip2   LDA  #0
1640          STA  &74
1650          JMP down1
1660  .down    JSR cursor
1670  .down1   LDA  #31
1680          JSR osasci
1690          LDA  #0
1700          JSR osasci
1710          LDA  #24
1720          JSR osasci
1730          LDA  &70
1740          CLC
1750          ADC  #8
1760          STA  &70
1770          LDA  &71
1780          ADC  #0
1790          STA  &71
1800          LDA  &70
1810          CLC
1820          ADC  #88
1830          STA  &72
1840          LDA  &71
1850          ADC  #0
1860          STA  &73
1870          JSR line
1880          JMP start
1890  .key4    CMP  #&8E    \ Check for
1900          BNE key5     \ shifted

```

```

1910          INC &71      \ cursor-down.
1920          JMP display
1930 .key5    CMP  #&8F     \ Check for
1940          BNE key6     \ shifted
1950          DEC &71      \ cursor-up.
1960          JMP display
1970 .key6    JMP key
1980
1990 .byte    STA &75      \ Display byte
2000          AND #&F0     \ in hex.
2010          LSR A
2020          LSR A
2030          LSR A
2040          LSR A
2050          JSR nibble
2060          LDA &75
2070          AND #&F
2080 .nibble  CLC          \ Display nibble
2090          ADC #48      \ in hex.
2100          CMP #58
2110          BCC print
2120          CLC
2130          ADC #7
2140 .print   JMP osasci
2150
2160 .line    LDA #&83     \ Display line
2170          JSR osasci   \ of table.
2180          LDA &73
2190          JSR byte
2200          LDA &72
2210          JSR byte
2220          LDA #32
2230          JSR osasci
2240          LDY #0
2250 .loop3   LDA #&82
2260          JSR osasci
2270          LDA (&72),Y
2280          JSR byte
2290          INY
2300          CPY #8
2310          BNE loop3
2320          LDA #13
2330          JMP osasci
2340

```

```
2350 .cursor1 LDA #31      \ Move text cursor
2360          JSR osasci   \ to position
2370          LDA &74     \ for editing
2380          ASL A        \ cursor.
2390          CLC
2400          ADC &74
2410          CLC
2420          ADC #6
2430          JSR osasci
2440          LDA #12
2450          JMP osasci
2460
2470 .cursor JSR cursor1  \ Remove editing
2480          LDA #&82     \ cursor.
2490          JMP osasci
2500 ]
2510 NEXT
2520 CALL monitor
```

INTERRUPTS

With a computer as sophisticated as the BBC Micro, which supports a large number of software-driven peripherals, there are a number of 'housekeeping' tasks the computer must perform regularly to keep these peripherals ready for the user. The processor is not, unfortunately, able to do two jobs at once so it must regularly stop what it's doing to check up on the peripherals. However, there is no point in the processor doing this unless a peripheral actually needs servicing.

To get around this problem the computer uses a system called *interrupts*. What happens is that there is a wire, connected to the processor, that is normally at a logic level of one (5 Volts). This wire is also connected to *all* the peripherals that may need servicing. When, say, the cassette system needs attention it pulls this wire down to logic level zero (0 volts). This interrupts the processor in what it's doing. The processor finishes the machine code command it was processing at the time the interrupt occurred and then pushes the contents of the program counter (high then low) on the stack, and then the status register. It then looks at two bytes at the end of the memory (&FFFE and &FFFF). These two bytes (low then high) make up the address of the operating system routine which handles interrupts.

There are two types of interrupts on the 6502 processor. These are IRQ (Interrupt ReQuest) and NMI (Non Maskable Interrupt). These are triggered by two separate wires on the processor. The most used form is the IRQ. When this occurs the processor looks at bit 2 of the status register—the interrupt disable flag. If this is *set* then it *ignores* the interrupt, otherwise it jumps to the address pointed to by

&FFFE (low) and &FFFF (high). In contrast, the NMI is not masked by the interrupt disable flag and so cannot be ignored. It jumps to the routine pointed to by &FFFA (low) and &FFFB (high).

Because the NMI is unstoppable it is only used for very important peripherals such as the disc system and the Econet interface which need fast service to function properly. All the other peripherals are on the IRQ. Inevitably, servicing their interrupts takes time. If you are prepared to ignore all the hardware that is interrupt-driven you can speed a program up quite noticeably by disabling interrupts. To do this you must set the interrupt disable flag in the status register with the SEI command. It is important to clear the flag again, when you have finished, by using the CLI command.

The operating system now has a chance to service the peripheral that generated the interrupt. However, before it can do this it must save the registers on the stack. This way the routine can reload them before returning execution to the main program. If this is not done then the main program will suddenly find its registers have changed and will probably crash. To save the registers the operating system uses the following commands:

```
PHA
TXA
PHA
TYA
PHA
```

When the operating system has finished servicing the interrupt it must return to the main program. First it must reload the registers.

```
PLA
TAY
PLA
TAX
PLA
```

Then it must use the command RTI. This reloads the status register and program counter from the stack and allows the processor to carry on from where it

left off before the interrupt occurred.

Next we need to look at the way the operating system handles an interrupt. It only knows that an interrupt has been generated somewhere in the computer. It doesn't know which piece of hardware has generated it. To find out it must look at each piece of hardware in turn until it finds which is the culprit (it is possible, though unlikely, that two or more devices may have generated interrupts simultaneously).

Luckily, each piece of hardware that can generate an interrupt has a register stored in the memory which contains a flag bit which indicates whether it has generated an interrupt or not. There are more details on each piece of hardware in *The Advanced User Guide*.

The devices which can cause interrupts on a BBC Micro are:

NMI

1 Mhz bus
Econet interface
Disc interface

IRQ

TUBE interface
1 Mhz bus
Cassette / RS423
System VIA

The system VIA is the most interesting to us, as this generates all the interrupts that keep the computer working normally.

Because a device must have an interrupt flag in it for the operating system to check, devices that don't have such a flag cannot directly generate interrupts. Instead their interrupt outputs are connected to some inputs on the system VIA. This has four inputs that can generate interrupts; and it has registers in it with flags for each input. The four devices in the 'Beeb' which can generate interrupts in this way are the light pen input on the analogue connector, the analogue to digital converter, the video controller and the keyboard.

We are most interested in the last two. The video controller generates an interrupt every time a vertical sync pulse is sent to the video monitor. This can be used for generating flicker-free graphics (see chapters 7 and 10). Here we will discuss the other interrupts and will also discuss events.

The system VIA

The keyboard generates an interrupt each time a key is pressed. When this happens the operating system looks to see which key has been pressed and updates the keyboard buffer. If you are writing a game and you don't need to use the GET command then you can disable this interrupt to speed up the game as this will not prevent you from using INKEY with a negative number.

There are also two timers in each VIA (user and system) which can be used either to generate an interrupt on a regular basis or to provide a single interrupt after a set amount of time.

The system VIA is memory mapped as sixteen registers at addresses &FE40 to &FE4F. These are regarded as registers zero to fifteen. (The user VIA is mapped similarly but at addresses &FE60 to &FE6F.) The VIAs are quite complicated to use and a lot of their functions are not particularly useful. There is a full description of them in *The Advanced User Guide*.

Let's first look at the four interrupt inputs for the system VIA. Its register 12 controls how these are used. For all the interrupts to work correctly this register must contain either 4 or 5. Normally it contains 4. This causes an interrupt at the end of the vertical sync pulse. By setting it to 5 the interrupt is caused at the beginning of the sync pulse. This is about two pixels earlier vertically. This may seem pretty pointless, but if you are using *FX19 for flicker-free graphics and the graphics flicker just at the top two pixels then this should cure it.

Normally you would not have to alter register 12.

Actual control of interrupts is done using registers 13 and 14. Register 14 is used to enable and disable

the various interrupts that the VIA can produce and can only be written to. When writing to register 14, if bit 7 is set, then a one in any other of the bit positions will enable the corresponding interrupt; if bit 7 is clear, then a one will disable the corresponding interrupt. This means that any interrupt can be enabled or disabled without affecting the other interrupts. Bits 0 to 6 in the system VIA's register 14 represent the following interrupts:

BIT	INTERRUPT
0	Keyboard
1	Vertical sync
2	Shift register
3	Light pen
4	A to D convertor
5	Timer 2
6	Timer 1

Register 13 contains the interrupt flags themselves for each part of the VIA. Each of bits 0 to 6 represents an interrupt as in register 14. If a bit is set this means that the relevant part of the VIA has caused an interrupt. Also, bit 7 is set if *any* one of the other bits is set. This provides a quick way for the processor to check if the VIA is responsible for the interrupt—it looks at bit 7 first.

These bits will not clear themselves so the first job the interrupt routine must do, once it has identified which interrupts have occurred, is to clear any bits that are set, ready for the next interrupt. To do this it must write to register 13 with the corresponding bits of the flags to be cleared, set. Note that writing a one to bit 7 of this register has no effect—in other words, to clear bit 7 you have to clear *all* the other bits.

There are also a number of jobs, such as updating the TIME clock, that the computer must do regularly. This is done using TIMER 1 in the system VIA. This is set to produce an interrupt every hundredth of a second. It is probably the most useful interrupt to us as it can enable us to do a bit of extra processing every hundredth of a second. This means that we can run two programs simultaneously so long as

one of them does not need much processing time and breaks down into convenient short sections which can be executed every hundredth of a second.

Interrupt-driven music

Now that we have seen a bit about how interrupts work we can look at an example. Because of the nature of interrupts they can, to a limited extent, be used to make the computer seemingly do two jobs at once. In this example we are going to make the computer play a tune, using interrupts. This will leave the computer free to do almost anything else (apart from using the sound port) in the meantime.

To make the program simple we will take a tune that can be played on channels one to three without envelopes. To make the program as 'transparent' to the user as possible we will not use any zero page addresses but will use variables stored directly after the program. The program itself can be placed in page ten of the memory. The data for the tune can be placed in page nine. We will need five variables: *time* which will count interrupts to produce a regular beat; *count* which will count the beats for an individual note; *point* which will point into the note table; and *tempx* and *tempy* for temporary storage of registers. The first three we can set to their initial values for the start of the tune. We will also need an eight-byte OSWORD command block for the SOUND command. While we are setting this up we can set some of the eight bytes that will not change throughout the program, so saving a few bytes of program.

```
.time    EQUB 1
.count   EQUB 1
.point   EQUB 0
.tempx   EQUB 0
.tempy   EQUB 0
.cblock  EQUD 0
          EQU D &FF0000
```

(90-150)

The main IRQ intercept routine will start at *irq* so first we need an initialisation routine to set up the IRQ vector.

We must first set the interrupt disable flag to prevent an interrupt occurring while we are changing the vector.

Then we must make a copy of the contents of the vector. This is so that when we have finished our interrupt work we can pass the interrupt on to the usual operating system routine.

Then we must reset the IRQ vector to point to our own routine.

Finally, we need to clear the interrupt disable flag and return.

```
.init    SEI
          LDA &204           \ Copy IRQ vector
          STA &230           \ into spare vector.
          LDA &205
          STA &231
          LDA #irq MOD256    \ Set IRQ vector
          STA &204           \ to irq.
          LDA #irq DIV256
          STA &205
          CLI
          RTS
```

(160-260)

Now we can write the main program. The first thing this must do is set the interrupt disable flag. This should stop any untimely interruptions. Next we must save the registers on the stack. The accumulator has already been stored at &FC for us by the operating system, so we need only save the X and Y registers.

```
.irq     SEI
          TXA
          PHA
          TYA
          PHA
```

(270-310)

Next we need to check that TIMER 1 is responsible for the interrupt. We examine bit 6 of register 13 of the system VIA. If this is set then TIMER 1 is responsible. We must not reset this flag as the operating system also wants a chance to service this interrupt. If TIMER 1 is not responsible then we can let the operating system cope with the interrupt. First we must reload the X and Y registers from the stack and then jump back to the normal IRQ routine in the operating system.

```
                LDA #&40
                BIT &FE40
                BNE irq1
.exit          PLA
                TAY
                PLA
                TAX
                JMP (&230)
```

(320-390)

We now have a routine that is called every hundredth of a second. However, we only want to change the notes of our tune every eight-hundredths of a second, otherwise the tune would be much too fast. To do this we decrement the variable *time* every hundredth of a second and, every time it reaches zero, reset it to eight and call the music routine.

```
.irq1         DEC time
                BNE exit
                LDA #8
                STA time
```

(400-430)

Now we have the problem that some notes are longer than others. If we have the length of the previous note in eight-hundredths of a second stored in *count* then we can decrement it each time until it reaches zero; at which point we can play the next note.

```
DEC count
BNE exit
```

(450-460)

Now we are almost ready to play a note. However, before we do this we must look at the way the notes are stored in the table. For this program they are stored four bytes per note. The first byte is the length of the note in eight-hundredths of a second and the other three bytes are the pitches of the three channels.

So the first thing we must do is to store the length of the note in *count*. The pointer into the table (which is less than 256 bytes long) is stored in *point*. This pointer counts in bytes so we must load it into the X register to use ABSOLUTE X addressing.

```
LDX point
LDA &900,X
STA count
```

Now we must play the three notes of the chord. We can do this with a loop using the Y register to count with.

```
LDY #3
```

(460-490)

Firstly we must increment X to point to the second byte of the entry. Then we must set up the OSWORD command block. The layout of the block for a SOUND command is that the first two bytes are the channel number, the next two bytes are the amplitude, the next two bytes are the pitch and the last two bytes are the duration.

Now we must set the channel number. We are also going to use the flush control so that each note wipes out the previous one. This is to make each note start during the interrupt. We will also make the duration 255 so that each note carries on until the next note is played. We have the channel number in

Y—we only have to add &10 to set the flush control. The high byte of the channel number is already zero.

```
.channel INX
      TYA
      ORA #&10
      STA cblock
```

(500-530)

Next we have the volume. For this tune we need some short rests to stop notes running into each other (a sort of staccato effect). For these notes the pitch number is zero. We must first set the volume to zero and then look at the pitch. If the pitch is zero then we have finished setting up the command block, otherwise we must set the volume to -15 (&FFF1 in two's complement) and set the pitch accordingly (again, the high byte of the pitch is already zero).

```
LDA #0
STA cblock+2
STA cblock+3
LDA &900,X
BEQ rest
LDA #&F1
STA cblock+2
LDA #&FF
STA cblock+3
LDA &900,X
STA cblock+4
```

(540-640)

Notice that the duration of the note is already set to 255.

Next we must save the X and Y registers and then set them to point to the command block. We must set the accumulator to seven for a SOUND command and call OSWORD.

```
.rest STX tempx
```

```
STY tempy
LDX #cblock MOD256
LDY #cblock DIV256
LDA #7
JSR &FFF1
```

(650-700)

Now we must reload the registers and if there is still a channel to be done we must go back and do it.

```
LDX tempx
LDY tempy
DEY
BNE channel
```

(710-740)

The X register now points to the fourth byte of the note in the table. By incrementing X it will point to the first byte of the next note and we can save it in *point*. If it has reached 168 then the whole tune has been played and we need to go back to the beginning by resetting *point* to zero. At this point we have finished with the interrupt routine and can pass control back to the operating system.

```
INX
STX point
CPX #168
BNE exit
LDX #0
STX point
JMP exit
```

(750-810)

We have finished the machine code now, so we only have to set up the actual tune. The easiest way to do this is to put it into DATA statements after the assembly code (lines 850-1050). For convenience we can set it up in lines of eight hexadecimal bytes run into a string. For example:

To place this data in the memory we need a few lines of BASIC. There are 21 lines of DATA for our tune so we need to read each one in as a string.

```
10 FORA%=0T020
20 READA$
```

Next we need to extract the eight bytes from A\$. These will be placed from &900 onwards. We can extract each byte using MID\$. Then we precede the string we have obtained with & and use EVAL to find its value.

```
30 FORB%=0T07
40 B%?(&900+A%*8)=EVAL("&" + MID$(A$,B%*2+1,2))
50 NEXT,
```

Notice that line 840 calls *init*.

The complete program with data looks like this:

```
10 FORA%=0T020
20 READA$
30 FORB%=0T07
40 B%?(&900+A%*8)=EVAL("&" + MID$(A$,B%*2+1,2))
50 NEXT,
60 FORpass%=0T02STEP2
70 P%=&A00
80 [ OPTpass%
90 .time EQUB 1
100 .count EQUB 1
110 .point EQUB 0
120 .tempx EQUB 0
130 .tempy EQUB 0
140 .cblock EQUQ 0
150 EQUQ &FF0000
160 .init SEI \ Set irq vector
170 LDA &204 \ to point to our
180 STA &230 \ routine.
190 LDA &205
200 STA &231
210 LDA #irq MOD256
```

```

220          STA &204
230          LDA #irq DIV256
240          STA &205
250          CLI
260          RTS
270 .irq     SEI          \ Main routine.
280          TXA
290          PHA
300          TYA
310          PHA
320          LDA #&40
330          BIT &FE4D
340          BNE irq1
350 .exit    PLA
360          TAY
370          PLA
380          TAX
390          JMP (&230)
400 .irq1    DEC time     \ Centisecond
410          BNE exit     \ clock trapped.
420          LDA #8
430          STA time
440          DEC count
450          BNE exit
460          LDX point
470          LDA &900,X
480          STA count
490          LDY #3
500 .channel INX         \ Set up each
510          TYA         \ channel.
520          ORA #&10
530          STA cblock
540          LDA #0
550          STA cblock+2
560          STA cblock+3
570          LDA &900,X
580          BEQ rest
590          LDA #&F1
600          STA cblock+2
610          LDA #&FF
620          STA cblock+3
630          LDA &900,X
640          STA cblock+4
650 .rest    STX tempx

```

```
660          STY tempy
670          LDX #cblock MOD256
680          LDY #cblock DIV256
690          LDA #7
700          JSR &FFF1
710          LDX tempx
720          LDY tempy
730          DEY
740          BNE channel
750          INX
760          STX point
770          CPX #168
780          BNE exit
790          LDX #0
800          STX point
810          JMP exit
820 ]
830 NEXT
840 CALLinit
850 DATA0B64544D01000000
860 DATA0364544D01000000
870 DATA0368544801000000
880 DATA0668544802000000
890 DATA0364584801000000
900 DATA06685C4802645C48
910 DATA035C483801000000
920 DATA0354483801000000
930 DATA0250402C0240402C
940 DATA0248402C0250402C
950 DATA0254402C025C402C
960 DATA0264402C0268402C
970 DATA0770645401000000
980 DATA0768504001000000
990 DATA0264544002545440
1000 DATA02685040025C5040
1010 DATA0370544001000000
1020 DATA03685C4801000000
1030 DATA0764544001000000
1040 DATA075C403801000000
1050 DATA0F54342401000000
```

Events

There is, however, a simpler way to use interrupts: an interrupt system called EVENTS. In this system ten common events that can occur are offered to the

user for processing. These events are slightly slower to respond than the interrupts. The events available are:

Event no.	Event
0	Output buffer empty.
1	Input buffer full.
2	Character entering input buffer.
3	ADC conversion complete.
4	Start of vertical sync.
5	Interval timer reaching zero.
6	ESCAPE has occurred.
7	RS423 error.
8	Econet event.
9	User event.

There is a complete description of events in *The Advanced User Guide*. If one of these events occurs, the operating system jumps to the subroutine pointed to by the vector at &220 and &221 (with a JSR command) with the event number in the accumulator. The X and Y registers may contain extra data according to the event.

Once an event routine has been set-up with the vector, the relevant event must be enabled. If more than one event is to be used, the event routine must test the accumulator first to check which event has occurred. If only one event has been enabled then this is not necessary. To enable an event use *FX14 with the number of the event.

Event routines must save all three registers, as with interrupts, and must not enable interrupts.

Probably the two most useful of these are events four and six. The vertical sync event (four) can either be used for flicker-free graphics (see chapters 7 and 10) or can be used to provide a regular interrupt every fiftieth of a second. For example, in our interrupt-driven sound program we could re-write the program in the following ways. The initialisation routine will need to be re-written.

```
160 .init      LDA #event MOD256
170           STA &220
```

```
180          LDA #event DIV256
190          STA &221
200          LDA #14
210          LDX #4
220          LDY #0
230          JMP &FFF4
```

We will need to delete lines 240 to 260. And the following lines must be changed in the main routine.

```
270 .event  PHA
320          DEC time
330          BEQ event1
```

Delete line 340

```
385          PLA
390          RTS
```

Delete lines 400 and 410

```
420 .event1  LDA #4
```

Notice that, as the vertical sync only occurs half as often as the centisecond interrupt, the value stored in *time* is halved.

The other event that is quite useful is the ESCAPE event. When this is enabled the normal action of ESCAPE is disabled—in fact, the ESCAPE key does absolutely nothing except generate an event. In a long machine code program such as a game this can be very useful, as it stops the ESCAPE key from having undesirable effects, but still enables it to be used to stop the game, etcetera.

In fact, because the relative simplicity of events makes them less versatile, they are not as useful as interrupts. For instance you can't use the interval timer in the system VIA using events. This timer is useful for accurate timing, as we shall see in chapters 7 and 10.

We will see in the next few chapters just how useful interrupts can be.

BRK Before we leave the subject of interrupts, there is one other machine code command we must discuss—BRK (or BReaK). Probably the closest equivalent to this in BASIC is the command STOP. BRK is always used with implied addressing.

When the processor comes across a BRK command it sets the Break flag in the status register and then carries on as if an IRQ has occurred. So, when the operating system handles an IRQ, it first has to check whether it is an IRQ or a BRK that has occurred. There is no machine code command that gives direct access to the Break flag, so at first sight this would seem to be a problem.

However, as with an IRQ, when the processor comes across a BRK command, it pushes the program counter then the status register, on the stack. Thus the operating system only has to pull the top byte from the stack. This will be the contents of the processor status register at the moment when the BRK or IRQ occurred. By testing bit 4 of this, the operating system can tell which of the two has occurred.

On the BBC Micro BRK commands are used for trapping errors. When a language ROM has found an error (for instance a syntax error), it has a BRK command followed by the error number, the error message, and a zero. We can use this in our machine code programs if they are called from BASIC.

We can run the following program and BASIC will assume that the error is an ordinary BASIC error and will behave accordingly.

```
10 ONERRORGOTO120
20 DIMmc%25
30 P%=mc%
40 [OPT2
50.error BRK
60 EQUB 255
70 EQUB "An error has occurred"
80 EQUB 0
90 ]
100 CALLerror
110 END
```

```
120 REPORT:PRINT" at line ";ERL
130 PRINT"Error number ";ERR
```

What happens is that when the operating system handles a BRK command it jumps to the Break vector at &202 and &203 (low, high). The A, X and Y registers are unchanged from when the BRK occurred. Also, the operating system pushes the status register from when the BRK occurred and the address of the next-but-one byte from the BRK command on the stack. This means that if the Break vector is set to point to an RTI command the processor returns to the next-byte-but-one from the BRK command, and carries on where it left off.

BASIC claims this vector and sets it to its own error handling routine. This routine pulls the top three bytes off the stack and discards them. Thus the BRK command has much the same effect as a jump to the error routine. It is this error routine that expects the error number and string directly after the BRK command.

Other languages may claim the BRK command for themselves simply by changing the contents of the Break vector.

Another example of the use of this vector is in intercepting errors in BASIC. The following program intercepts all BASIC errors. It must first save the two registers it uses so that the routine is 'transparent'. It then looks at the byte after the BRK command. It can do this because the operating system makes a copy of this address in the zero page locations &FD and &FE. If this error is an 'escape' (error number 17) then it ignores it, otherwise it prints a message before returning to the normal BASIC error routine.

```
10 FORpass%=0TO2STEP2
20 P%=&A00
30 [OPTpass%
40.brk PHA
50 TYA
60 PHA
70 LDY #0
80 LDA (&FD),Y
```

```
90          CMP #17
100         BEQ exit
110.loop    LDA fool,Y
120         JSR &FFE3
130         INY
140         CPY #13
150         BNE loop
160.exit    PLA
170         TAY
180         PLA
190         JMP (&230)
200.fool    EQU $13+"Silly Billy!"
210 ]
220 NEXT
230 ?&230=?&202
240 ?&231=?&203
250 ?&202=0
260 ?&203=10
```

A FEW WAYS TO PROTECT YOUR PROGRAMS

Piracy is a problem that is worrying many software houses these days. It is impossible to produce a commercial program that cannot be copied—however clever the protection is, someone will find a way around it. However, it is possible to make it very difficult for anyone to copy a program. Very few people have the skill and patience to copy a program which has been properly protected. Here you will find a few of the many techniques that can be used to make copying difficult.

It is possible to write a program in such a way that, once it is run, it cannot be stopped. To do this we need to disable the ESCAPE key and the BREAK key. The ESCAPE key is easy to disable but the BREAK key cannot be disabled completely. It is, however, possible to make the computer clear the memory when the BREAK key is pressed. The Operating System conveniently provides a *FX call to deal with both the ESCAPE and BREAK keys. This is *FX200. This is used to change a flag byte within which only bits 0 and 1 do anything. If bit 0 is set then the ESCAPE key is completely disabled, and if bit 1 is set the BREAK key causes the memory to be cleared. Thus, by placing the command *FX200,3 at the beginning of a program, it is impossible to get out of the program without the memory being cleared. (We will see in chapter 6 how to intercept the BREAK key.)

Locked tape files

This is all very well, but a program can still be loaded and then saved. We need a way of stopping people doing this. Acorn have kindly placed a protection system in the BBC Model B Operating System for tape users. This system produces a file on tape which is 'locked'. If you try to *LOAD such a file you will get the message 'file locked'. In fact, the only command the operating system will allow you to use on a locked file is *RUN. This means that only machine code programs can be locked.

A locked file is produced by setting a particular bit in the header of each block. This tells the operating system that the file is locked. The operating system does not, unfortunately, provide a command for saving locked files, so we need a program that will do this. At first it would appear that we need to write a complete save routine to do this. There is, however, a beautifully simple way.

While the operating system is saving a file it keeps a copy of the header for each block in locations &3B2-&3D0. The bit we are interested in is bit 0 of location &3CA. Because the tape hardware is interrupt driven, not only does an interrupt occur every time a byte is saved to tape, but also, the centisecond clock is left running. It is very simple to redirect the main interrupt vector so that as each byte is saved bit 0 of &3CA is set. The following program does just this. Notice that the machine code is stored in *zero* page where it can't get in the way. Once this program is run all files saved will be locked until BREAK is pressed.

```
10 P%=&50
20 [OPT3
30 .irq PHA
40     LDA &3CA
50     ORA #1
60     STA &3CA
70     PLA
80     JMP (&230)
90 .init SEI
100    LDA &204
110    STA &230
120    LDA &205
```

```
130      STA &231
140      LDA #irq MOD256
150      STA &204
160      LDA #irq DIV256
170      STA &205
180      CLI
190      RTS
200 ]
210 CALLinit
```

We now would appear to have an uncopyable program which can only be *RUN and, once run, cannot be broken into. Of course, it would be naive to think even this system infallible. It is possible to crack a locked file but that technique will not be revealed here.

Unlistable programs

It is sometimes useful to be able to write a BASIC program which cannot be listed. This is done using ASCII code 21 which disables the VDU drivers. When this has been sent to the VDU drivers the screen ignores everything that is sent to it until a VDU6 command is used to re-enable the VDU drivers again. By placing CHR\$21 and CHR\$6 codes in a listing, part or all of a program can be made unlistable.

Unfortunately, these codes cannot just be typed into a program listing. If, however, two character codes which are not used anywhere else in a program are used—codes such as the 'curly' brackets—then it is quite easy to write a short program that will convert these into CHR\$21 and CHR\$6 codes. So that the program will run properly these characters should be placed in REM statements. Other characters can be used, such as delete (CHR\$127); or cursor controls, such as line-feed. The obvious way to use this is to disable the whole listing, but it is often much more effective to use it to remove particular lines without which the program appears to do something completely different. This system can be used to great effect as the first part of a game which prints instructions and then chains or *RUNs the next section. Try typing-in this example program:

```

10 REM @*****{
20 REM }*      *{
30 REM }*   This program is copyright *{
40 REM }*      *{
50 REM }*   It is illegal to copy it. *{
60 REM }*      *{
70 REM }*****{
80 REM REST OF PROGRAM
90 PRINT"THIS PROGRAM WILL STILL RUN"
100 GOTO90
110 REM}

```

We want to change @ (line 10) to a clear screen (code 12); { to CHR\$21; and } to CHR\$6 (lines 20 to 70 and 110). To do this you must first type the commands:

```

PAGE=&2000
NEW

```

Then you must type in and run the following program. Disc users will need to change the setting of A% at line 10 to &1900.

```

10 A%=&E00
20 IFA%?1=&FF END
30 A%=A%+3
40 A%=A%+1:IF?A%=13 THEN20 ELSE IF?A%<>&F4
THEN40
50 REPEAT A%=A%+1:IF?A%=ASC"@ " ?A%=12
60 IF?A%=ASC")" ?A%=6
70 IF?A%=ASC"{" ?A%=21
80 UNTIL?A%=13:GOTO20

```

Notice that in line 50 the REPEAT command does not need a : after it. This is never needed after a REPEAT statement.

Now set PAGE back to its usual value and try listing the program. Note that the program still runs normally.

This program first finds the start of each line of the program then looks for the token (&F4) for a REM statement. It then searches the rest of the line for the characters we want to replace. This way the

program won't accidentally alter line numbers or lines of BASIC.

Alternatively, you can use a machine code monitor (such as the one in chapter 4) to look directly at and modify the relevant characters in the BASIC coding.

A very effective (and far more subtle) way of using this technique is to double-bluff the pirate. For instance, say that in the BASIC loader for a machine code game the last two commands are:

```
*LOAD game 3000  
CALL &3082
```

The pirate will look at this and will be able to examine the machine code. If we can arrange for him not to know the load and execution addresses then his task is much harder. Even better, if we can convince him that the load and execution addresses are, say, &2800 and &293A respectively. . . .

The way to do this is to put a dummy set of commands at the end of the program and then conceal the real ones. The original program ending would look like this:

```
320 REM * load machine code *{  
322 *LOAD game 3000  
324 CALL &3082  
326 REM }  
330 *LOAD game 2800  
340 CALL &293A
```

Try typing this in and using the alteration program as before, then listing the program.

Note that the line numbers that the pirate will see are in steps of ten, leaving no clue to the missing lines.

One word of warning before you use this technique. There is a major, very annoying bug in the operating system! When the VDU drivers have been disabled with VDU21 any carriage returns sent to the VDU drivers are sent to the printer: even if the printer hasn't been enabled—even if your printer has been turned off—even if you don't have

a printer! This irritating quirk means that every time you list your program the printer spits paper at you! Also, if your printer is not turned on, these carriage returns (and line-feeds if you have used *FX6,10) accumulate in the printer buffer. If the number of these reaches 63 the whole computer seizes up until either ESCAPE or BREAK is pressed. So don't fill your programs full of CHR\$21's.

If you are writing a program with the VDU21 command in it you can get around this by using the command *FX3,64 before using VDU21. This disables the printer driver completely except for characters sent using the VDU1,x command.

Disc tricks

For those of you with disc drives the list of fiendish tricks you can play on the pirate is endless. These tricks all rely on knowing how to access the blocks on a disc directly, using an OSWORD call. The DFS adds a series of extra calls to the standard list of OSWORD calls. We are going to use one of these—OSWORD &7F. This routine saves or loads a section, or all, of a track.

On entry to this routine the X and Y registers must point to a parameter block (Y high). This parameter block consists of 10 bytes and should be laid out as follows:

XY+ 0	Drive number.
XY+ 1	
to	Load / Save address.
XY+ 4	
XY+ 5	Number of parameters (3)
XY+ 6	Command (&53 for load, &4B for save).
XY+ 7	Track number.
XY+ 8	Sector number.
XY+ 9	High nibble: sector length in 128- byte groups. Low nibble: number of sectors to be loaded/saved.
XY+10	Error number (0 if no error).

For example, if we wanted to read the contents of the block at track 10 sector 5 we would need to reserve 256 bytes in the memory to store the block.

Then we would need to set up a parameter block. The first byte of the parameter block would be zero and the next four would point to the reserved block of memory (the top two bytes would be set to zero). The number of parameters refers to the number of parameters after the command byte that are sent to the routine (i.e. not including the error byte) and so would be three. The command would be &53. The track number would be 10 and the sector number 5. The last (ninth) byte is more complicated. The size of a block on a standard DFS disc is 256 bytes, so the high nibble of the parameter byte needs to be 2. We only want to load one block, so the low nibble is 1, i.e. the ninth byte is &21.

For protection against piracy there are a number of things we can do with this. The first is to save files with names that include control codes. As with the unlistable programs, we can apparently totally eliminate files from the catalogue and yet still load them. This means that unless the pirate tries every name he can think of (quite a long job!) only a person who knows the filename can load the file. This is not much use if you want to sell the program, but if you write a well-protected loader which then chains the main program then only the loader's filename need appear on the catalogue.

If you have the old DFS then saving control codes in filenames is easy. For example, to save a file that appears on the catalogue as TEST but actually has a different filename, try:

```
SAVE "TESS|HT"
```

The problem with this is that the catalogue and compact routines use the length of the filename to set out the catalogue on the screen. So, if you are not careful, filenames will tend to be shifted left if they are printed on the same line as, but further right than, a doctored filename. This is where the subtle approach is not necessarily the best. Very good effects can be obtained by titling the disc with a clear screen, a suitable message, and a CHR\$21! You can then add a CHR\$6 to the end of the last file on the catalogue. If your disc boots then you don't even

have to let the user catalogue the disc at all!

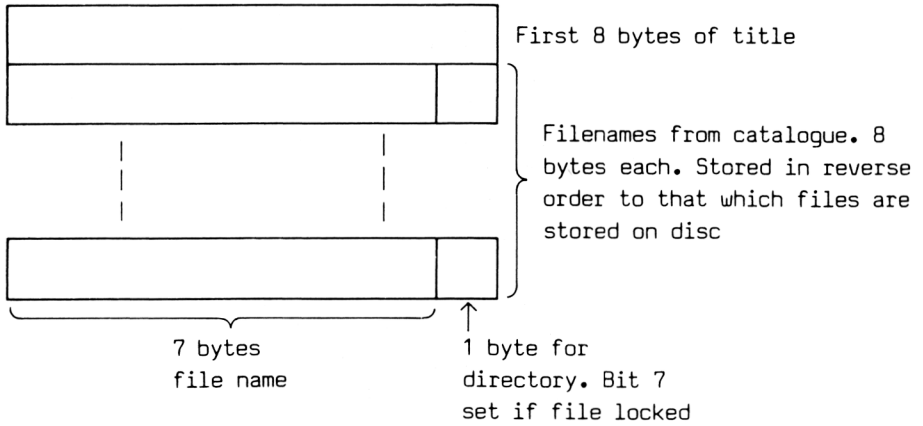
If you don't have the old DFS you will need to load the first block on the disc (track zero, sector zero) into the memory, alter it accordingly, and save it back again—a somewhat more cumbersome method.

However, all this is child's play next to what can be achieved. It is not very useful to stop the pirate loading the program and looking at it if he can just 'backup' to a new disc. Here's how to stop him:

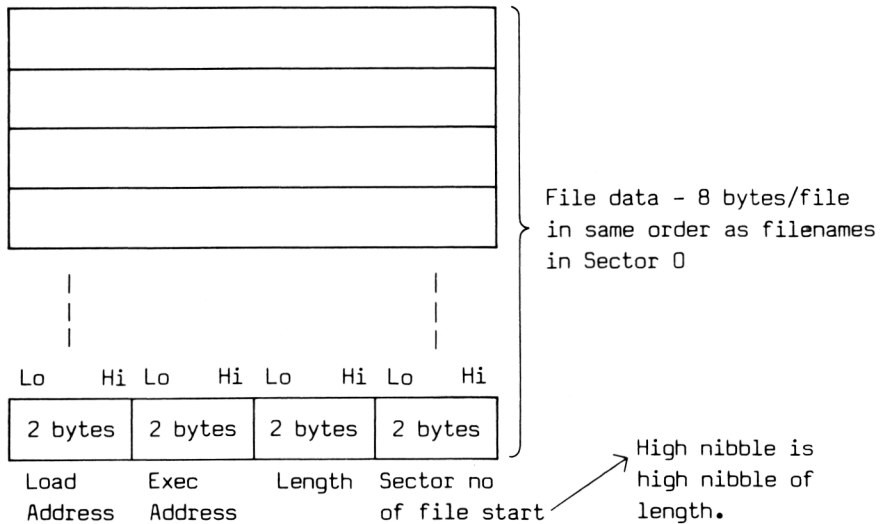
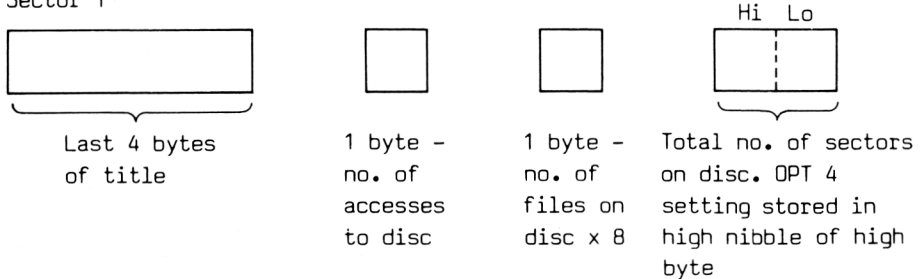
The second block on the disc contains the load and execution addresses, the lengths, and the locations on the disc, of all the files. It also contains eight bytes at the very start of the block. The first four of these are the last four bytes of the title (the first eight bytes of the title are the first eight bytes of block zero). Then comes one byte which stores the number of times the disc has been written to since formatting (this number appears in brackets after the title in the catalogue). Then comes a byte which gives the number of files in the catalogue, times eight. And finally, two bytes that give the total number of blocks on the disc. These are stored high byte first (yes, this is unusual). Also, the high nibble of the high byte is used to store the OPT number.

You may wonder of what use all this is. The answer is that we can set the total number of blocks on the disc to zero! This may seem foolish but it doesn't affect the way the DFS loads from disc. It only has an effect when you start saving on the disc or when you back-up the disc. When backing-up, the computer copies on to the new disc the number of blocks specified in block one of the source disc. This means that, by setting this to zero, the back-up command will copy precisely no blocks. If you want to be even more cruel to the poor, defenceless pirate, you could arrange for just enough blocks to be copied to copy the loader from disc but not the main program. The loader could then check a block at the end of the disc to see if it contains a specific string—which, of course, you will have put there on the original disc; and, if it is not there, then it would print a suitable message, such as 'This is an illegal copy' and crash!

Track 0
Sector 0



Sector 1



Note that if the length has been set to zero you can't save any more programs on the disc. It also fools certain utilities' commands for looking at the disc directly—pirates occasionally use this for breaking into discs.

A useful side effect is that *COPYing all the files on the disc will not copy the string at the end and so the copy won't work. This leads me to another suggestion for disc protection: have a well-protected loader which loads the main program off the disc directly using OSWORD &7F. This way you need not even put the main program on the catalogue. You might even like to store the program on disc in some personal encryption code!

Another interesting point is that, if you add 128 to all the bytes that make up a filename in block zero, that filename will promptly vanish from the catalogue completely! It will, however, still load and run perfectly normally—if you know the filename!

Yet another thing that you can do if you have a utility such as DISC DOCTOR which allows you to format sections of a disc, is take an unformatted disc and format it leaving some tracks unformatted. If the pirate tries to make a back-up of this he will get a load error! To cover these tracks set up a dummy file in the catalogue that occupies the space.

There are plenty of other ways of confounding the pirate. You could try coding (that is, encrypting) your program and placing a decoding routine at the start. This won't stop a determined pirate as he can deduce the code from the decoding program, but it will slow him down. In fact, you will find it very difficult to stop a determined pirate. You can only protect against the person who casually copies programs. By combining a large number of protection methods you can make the pirate's job difficult enough so that he will think twice before attempting anything. What you must decide is whether all this is worth it.

A companion book in the *Master Guide* series, *Mastering the Disc Drive* (BBC Publications, 1985) goes into the disc system in great detail.

THE KEYBOARD

At first sight there is not a lot that can be said about the keyboard except that it is the most important means of input the computer has.

The first thing that must be said is that programs of a 'professional' standard must be fool-proof. Probably the most common problem with commercial software is that particular keys on the keyboard have not been disabled properly. As a general rule, at any point in a program *all* keys except the ones that can be used legally should be disabled. This applies in BASIC and in machine code.

With this in mind we should take a look at all the keys and the methods of disabling them.

When you need to input data from the keyboard in a program you should check it to ensure that it is made up of legal key presses. In BASIC the easiest way to do this is to compare each character of the string with a string containing all the legal characters. This can be done with the command INSTR.

For example, suppose you want the user to input his first name. You want letters, upper or lower case, and nothing else.

The BASIC to check for this would be:

```
1000 CLS
1010 INPUT"Your first name";A$
1020 E%=0
1030 FORA%=1 TO LENA$
1040 IF INSTR("ABCDEFGHIJKLMN OPQRSTUVWXYZabcde
           fghijklmnopqrstuvwxyz",MID$(A$,A%,1))=0
           THEN E%=1
1050 NEXT
1060 IFE% THEN 1000
1070 PRINTA$
```

A BASIC input routine

This method of solving the problem still allows the user to type the wrong letters in the first place and a name of up to 250 characters, which would be a little silly! Better to write an input function which uses the GET command and checks each character as it is typed. If a character (such as a %, say) is illegally typed it ignores it and waits for another key to be pressed. Also, a maximum number of characters can be imposed. If this is done it is a good idea to set out a row of full stops, over which the user types, to show him how much he is allowed to type.

```
10 CLS:VDU23,1,0;0;0;0;
20 PRINT"Your first name?":A$=FNinput
   (20,"ABCDEFGHJKLMNPOQRSTUVWXYZabc
   defghijklmnopqrstuvwxyz")
30 PRINTA$
40 END
```

```
1000 DEFFNinput(N%,I$):LOCALA$,G$
1010 PRINTSTRING$(N%,".");STRING$(N%,CHR$8);
1020 A$="":VDU23,1,1;0;0;0;
1030 G$=GET$:IFG$=CHR$13VDU13,10,23,1,0;0;0;0;
   :=A$
1040 IFG$<>CHR$127THEN1070
1050 IFLENA$=OVDU7:GOTO1030

1060 VDU8,46,8:A$=LEFT$(A$,LENA$-1):GOTO1030
1070 IFINSTR(I$,G$)=OVDU7:GOTO1030
1080 IFLENA$=N%THENVDU7:GOTO1030
1090 A$=A$+G$:PRINTG$;:GOTO1030
```

Note also that this program only turns the cursor on (line 1020) when an input is expected. This is good practice as it helps to give the user a clue as to when he is expected to type something.

A machine code input routine

A like method can be used from machine code. Here we need a place to put the string. As BASIC is not being used we can use the space taken up by the BASIC string input buffer (appropriately enough). This is the whole of page seven of the memory.

Let's specify that the routine is entered with the maximum length of the string in X. We also need a string containing all the legal characters. This we can place at the end of the machine code program with the EQU S command. We can then say that the routine must be entered with the length of this string minus one in Y. This way we can allow different amounts of this string to be legal by changing Y. For instance, if you wanted to use the routine twice—the first time allowing the letters A to Z and the second time allowing only the letters A to F—then you could set out the legal string as 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'. The first time you called the routine, Y would be 25 to allow all the string to be legal, the second time, Y would be 5 so as to only allow the first six letters of the string to be legal.

The routine will exit with the string stored from &700 onwards, and followed by a carriage return. Thus the example would be:

```
10 PROCass
20 CLS:VDU23,1,0;0;0;0;
30 PRINT"Your first name?"
40 X%=20:Y%=51:CALLinput
50 PRINT$&700:END
```

Now we must write the routine.

The first job is to store the contents of the X register at &70 and the Y register at &72 temporarily.

```
1000 DEFPROCass
1010 DIMmc%250
1020 FORpass%=0TO2STEP2
1030 P%=mc%
1040 [OPTpass%
1050 .input STY &72
1060 STX &70
```

The next job is to print the row of dots over which the user will type. The number is already conveniently in the X register for us.

```
1070 .loop1   LDA #46
1080          JSR &FFE3
1090          DEX
1100          BNE loop1
```

Next we have to move back the same number of spaces to allow the user to start typing over the top of the dots. This time the X register will have to be reloaded from &70.

```
1110          LDX &70
1120 .loop2   LDA #8
1130          JSR &FFE3
1140          DEX
1150          BNE loop2
```

While we are at it we need a variable to count how many characters the user has typed in. This we can store at &71; it will initially need to be zero. As we have just finished a loop in X, the X register will contain zero; so we can store this at &71.

Then we must turn on the cursor with
VDU23,1,1,0;0;0;

```
1160          STX &71
1170          LDA #23
1180          JSR &FFE3
1190          LDA #1
1200          JSR &FFE3
1210          JSR &FFE3
1220          LDX #7
1230 .loop3   LDA #0
1240          JSR &FFE3
1250          DEX
1260          BNE loop3
```

Now we are ready to get a key from the keyboard using OSRDCH. If this routine exits with the carry flag set then the ESCAPE key has been pressed and we need to acknowledge this and exit the routine with a null string stored at &700.

As we do this we will need to put a carriage return at the end of the string and turn the cursor off.

This part of the program we can use as an exit once we have obtained a valid string, so we must give it the label *exit*.

```
1270 .key      JSR &FFE0
1280          BCC noerror
1290          LDA #&7E
1300          JSR &FFF4
1310          LDA #0
1320          STA &71
1330 .exit     LDX &71
1340          LDA #13
1350          STA &700,X
1360          LDA #13
1370          JSR &FFE3
1380          LDA #23
1390          JSR &FFE3
1400          LDA #1
1410          JSR &FFE3
1420          LDX #8
1430 .loop4    LDA #0
1440          JSR &FFE3
1450          DEX
1460          BNE loop4
1470          RTS
```

Next we must check to see if the key pressed is the RETURN key. If so, then we can branch to *exit*.

```
1480 .noerror  CMP #13
1490          BEQ exit
```

Next we check for DELETE. If this has been pressed then we check whether there is any string to be deleted. If &71 is zero then there is no string so we output a 'beep' before jumping back to *key*.

```
1500          CMP #127
1510          BNE notdel
1520          LDA &71
1530          BNE del
1540 .error     LDA #7
1550          JSR &FFE3
1560          JMP key
```

If there is something to delete then we need to go back a space, print a dot and back-space again to leave the cursor over the dot. We also need to decrement the length of the string stored at &71.

```
1570 .del      LDA #8
1580          JSR &FFE3
1590          LDA #46
1600          JSR &FFE3
1610          LDA #8
1620          JSR &FFE3
1630          DEC &71
1640          JMP key
```

If DELETE is not pressed then we need to check for a legal key. The string of legal characters starts at *legstr* and the length of this string minus one is stored at &72. We need to compare all the characters in this string with the accumulator. We can do this with a loop in X, carefully preserving A throughout the loop. If the contents of the accumulator match with one character of the string then the key is legal; otherwise, if we get to the end of the string without finding a match then we must cause a 'beep' and go back for another key.

```
1650 .notdel   LDX &72
1660 .loop5    CMP legstr,X
1670          BEQ legal
1680          DEX
1690          BPL loop5
1700          JMP error
```

If the key is legal then we must check that there is still space left to place this key. If the length of the string has already reached the maximum allowed length then we must branch to *error*. If not then we can store the character at the relevant place in page 7, increment the length of the string, print the character on the screen and go back for the next key.

```
1710 .legal    LDX &71
1720          CPX &70
1730          BEQ error
```

```

1740          STA &700,X
1750          INC &71
1760          JSR &FFE3
1770          JMP key
1780 .legstr EQU$ "ABCDEFGHIJKLMNQPQRS
          TUVwXYZabcdefg hijk1
          mnopqrstuvwxyz"
1790 ]:NEXT
1800 ENDPROC

```

We have now dealt with all the standard ASCII keys. The next thing to look at is the cursor keys. If a GET, INPUT, or INKEY (with a positive parameter) command is used during a program then the cursor keys become enabled. Pressing them will cause the copy cursor to move around the screen. This also leaves a block cursor on the screen. In most cases this is not desirable. There is an easy way to overcome this problem but not many people seem to even realise that there is a problem. For example, many games leave the cursor keys enabled—pressing them during the game will leave 'flying blobs' on the screen.

The way around the problem is to set the cursor keys to generate ASCII codes with the command *FX4,1. This way they can be used as ordinary keys. This command also causes the COPY key to generate an ASCII code.

The next keys we need to look at are the SHIFT LOCK and CAPS LOCK keys. These keys cannot be disabled easily but the state of the keyboard (and the LEDs) can be changed from software. This is done using a *FX202 call. This command needs one number after it. If bit 4 of this number is zero then the CAPS LOCK is engaged. If bit 5 is zero then the SHIFT LOCK is engaged. If bit 7 is set then the shift key's action is reversed.

For example, if *FX202,160 were used this would set the keyboard so that it would normally produce capitals and numbers, etcetera, but with the shift key pressed it would produce lower case and the exclamation mark, and so on.

This command also changes the state of the keyboard LEDs. An example of where it could be used

is in a word processor to turn the CAPS LOCK and SHIFT LOCK off at the beginning of the program.

The next key we must look at is the ESCAPE key. In machine code this key has little effect until OSRDCH is called. In this case the escape condition must be acknowledged with an OSBYTE &7E call. If this is done then the ESCAPE key is effectively disabled. However, if a more complete form of disablement is needed, say for a BASIC program, then *FX229,1 should be used. This simply causes the ESCAPE key to generate ASCII code 27. Another useful trick is that any key on the keyboard can be made the ESCAPE key. This is done by using the command *FX220 followed by the ASCII code for the key. For instance, if you wanted <CTRL @> to be the ESCAPE key then you would use the command *FX220,0.

One final method is to use *FX200. This has two functions. If bit 0 of the byte following it is 1 then ESCAPE completely disabled (it won't even generate an ASCII code) and if bit 1 is set then the entire contents of the memory will be cleared the next time the BREAK key is pressed! Even <CTRL BREAK> cannot get around this command. This is very useful for protecting programs as it means that once a program has been run it is impossible to get out of it again without losing the program (see chapter 5).

The BREAK key

The BREAK key is probably the most difficult key to disable. In fact, it is impossible to disable it. However, it can be intercepted. Many programs define the BREAK key like a soft key to produce a string that runs the program. This means that if you accidentally press BREAK in the middle of typing in a letter on your word processor, you won't lose your text. However, this doesn't solve the problem of <CTRL BREAK>. It would be nice if we could stop the computer every time the BREAK key is pressed and check whether a <CTRL BREAK> has occurred. If so, we could then convince the computer that it is imagining things and that the BREAK was really a normal one! Well, here's how to do it.

When a BREAK occurs, the operating system looks at location &287. If &287 contains zero then it carries on as usual; and, if the machine has just been turned on then it must be a zero! However, if this byte contains a machine code JUMP instruction (&4C) instead, then it will jump to the address pointed to by the contents of locations &288 and &289. In fact, it does this twice after a BREAK occurs. The first time the carry flag is clear, and this occurs before the message 'BBC Computer' appears; the second time is after this message appears but this time with the carry flag set. Locations &287 to &289 can be set using *FX247 to *FX249.

Those of you with fevered imaginations will already have seen some of the possibilities this opens up. For example, it is possible to change the 'BBC Computer' message, which is normally printed, to something completely different. For those of you who like the idea, here is the program:

```
10 FORA%=0T03STEP3
20 P%=&900
30 [OPTA%
40.break      BCC exit
50            LDX #0
60.loop       LDA string,X
70            JSR &FFE3
80            INX
90            CPX #24
100           BNE loop
110.exit      RTS
120.string    EQU$ CHR$12+"Beware of the
              hacker!"+CHR$13+CHR$13

130]:NEXT
140 *FX247,76
150 *FX248,0
160 *FX249,9
```

Notice that this is put in the cassette output buffer (&900). This is totally safe for disc users but cassette users will find that, if they save a program, the next time they press BREAK the computer will crash.

However, the object of all this was to disable <CTRL BREAK>. When a break occurs one of the

first things the computer does is to check whether it is a soft reset, a hard reset or a power-on reset. When it has done this it sets up a variable at &28D. This has the value zero for a soft reset, one for a power-on reset and two for a hard reset. By changing the contents of &28D to zero in our break intercept routine we will fool the computer into believing that a soft reset has occurred. Unfortunately, before we get to do this the computer has already reset the clock and cleared the function keys. However, if we set &28D to zero in the *first* intercept we can redefine the break key to do whatever we want in the second intercept. Here is an example program:

```
10 FORA%=0TO3STEP3
20 P%=&900
30 [OPTA%
40.break      BCS second
50            LDA #0
60            STA &28D
70            RTS
80.second     LDX #string MOD256
90            LDY #string DIV256
100           JSR &FFF7
110           RTS
120.string    EQU"*KEY10OLD|MRUN|M"+CHR$13
130 ]:NEXT
140 *FX247,76
150 *FX248,0
160 *FX249,9
```

This clears &28D during the first intercept and uses OSCLI to perform a *KEY10 command during the second intercept.

Once this is run neither BREAK nor <CTRL BREAK> can stop the computer from 'olding' and running the current program!

GENERAL GRAPHICS

The graphics system on the BBC Micro is based on two chips. These are the 6845 Cathode Ray Tube Controller (CRTC for short) and the Video ULA. Between them, these two chips are responsible for the flexibility that allows the BBC Micro to have eight different screen modes.

To understand how to use these chips we must first look at the way in which a TV monitor works. Inside a TV is a tube with the screen at one end of it. In this tube a narrow beam of electrons is fired at the screen. The screen is covered on the inside with a substance which glows where the beam hits it. The beam can obviously only illuminate one dot on the screen at once and yet we need large amounts of screen to be lit, seemingly continuously. What happens is that the beam is scanned across the screen in a series of horizontal lines from left to right starting at the top and working down. At the same time the beam is switched on and off to produce light and dark areas on the screen. The beam completes one vertical scan of the screen every fiftieth of a second, so the eye is fooled into seeing a coherent picture.

The graphics registers

The 6845 CRTC contains 18 registers that we can use. To write to them, the number of the register we want to access (from 0 to 17) must be placed at address &FE00 and the data can then be written through location &FE01. The legal way to write to a register is through the operating system, using VDU23. The format is like this:

```
VDU23,0,reg_no,data;0;0;0
```

There is no way to read from any register. There is a complete description of all these registers in *The Advanced User Guide*. For the purposes of this book we will only look at some of the more useful registers.

Register 1 gives the total number of displayed characters per line.

However, although we would assume that in Mode 2, say, it would contain 20, this is not so. The 6845 CRTC was not designed to be used for high-resolution graphics—it was originally designed as a straightforward text VDU controller, for text like that of Mode 7, with only one byte needed to store each character, though it is very simple to use it for black-and-white graphics where each character takes up eight bytes. So to use it as the heart of a colour display as required by the BBC Micro, it must be made to address and fetch the right number of bytes of data for each text character in that mode. For Mode 2 each text character takes up all of 32 bytes—not the eight bytes that a normal black-and-white display would need to store a character. The result is that for each text character on the Mode 2 colour screen the 6845 needs to fetch four of its eight-byte characters. This means that, in the 20-character Mode 2, the CRTC is made to think that there are 80 of *its* characters per line. Thus, for Modes 0 to 3, Register 1 contains 80, and for Modes 4 to 7 it contains 40.

By changing Register 1 you can make narrower screens. For instance, suppose you have a game that needs Mode 4 but is very long. If you are prepared to only use 256 pixels across then you can set Register 1 to 32. This will mean that the screen only takes up 8K instead of 10K. This technique is used by Acornsoft to fit *Elite* into the computer.

Register 2 contains the position of the horizontal sync pulse.

This is effectively used to position the screen on your monitor, left and right. In different modes it contains these values:

Mode	0	1	2	3	4	5	6	7
Register 2	98	98	98	98	49	49	49	51

If you are using the narrow 8K Mode 4 you will need to set this register to 45 instead of 49.

Register 6 is the vertical equivalent of Register 1.

It gives the number of vertical text lines to a screen. Its normal values for the modes are:

Mode	0	1	2	3	4	5	6	7
Register 6	32	32	32	25	32	32	25	25

This can be used together with Register 1 to produce smaller modes.

Register 7 contains the vertical sync position.

It is used by the *TV command to move the screen up and down to accommodate different monitors. As before, for smaller modes you may have to change this register.

Registers 12 and 13 give the start of screen address in RAM.

Notice that Register 12 is high and Register 13 is low! These two registers form the address of the top-left hand corner of the screen in RAM, but *divided by eight*. For example, in Mode 2, where the top left-hand corner is stored at &3000, these registers contain &600. By changing these registers you can make any part of the RAM the screen. Also, the registers can be used for scrolling. If the registers are set so that there is not enough addressed RAM after the top left-hand corner to display a complete screen—that is, the address of the bottom of the screen will occur after &7FFF—then the hardware wraps the screen back to before the top left-hand corner.

For example, if in Mode 2 we set Registers 12 and 13 to &B00, i.e. pointing to &5800—exactly half way down the screen—then, when the top half of the screen has been displayed the VDU will go back to &3000 to display the bottom half. This is how scrolling is done in all the modes, as this saves moving

large amounts of memory around. The point in the memory to which the screen scrolls back to is normally dependent on the mode and is set to the normal top-of-screen address. However, this is set as part of changing mode through the system VIA.

Bits 0 to 3 of port B on the system VIA are used as an addressable latch to control several functions of the BBC Micro. We are interested in controlling the scroll wrap-around. This control is achieved by altering the settings of two bits. The explanation of these is a bit (sic) complicated, so I shall just give the four relevant commands:

```
?&FE40 = &4  clears low bit.  
?&FE40 = &C  sets low bit.  
?&FE40 = &5  clears high bit.  
?&FE40 = &D  sets high bit.
```

These two bits in their four combinations set the start of screen RAM for scrolling, as follows:

high	low	address
0	0	&4000
0	1	&6000
1	0	&5800
1	1	&3000

So we now have all we need to set up an 8K 'narrow' Mode 4.

```
10 MODE4  
20 VDU23,0,1,32;0;0;0  
30 VDU23,0,2,45;0;0;0  
40 VDU23,0,12,12;0;0;0  
50 ?&FE40=&5: ?&FE40=&C  
60 ?&34E=&60: ?&351=&60  
70 VDU30  
80 HIMEM=&6000
```

This first adjusts Register 1 to set the number of characters per line to 32 (line 20). Line 30 then adjusts Register 2 to shift the whole screen right four characters to centre it. Next the top-of-screen is set to &6000 (Line 40) and the scrolling is set to wrap

around to &6000 (line 50). Finally, printing to the screen is set to start at &6000 (line 60) by setting the operating system's top-of-screen variables suitably and homing the cursor (line 70). We can now claim the extra 2K of memory by moving HIMEM up to &6000 (line 80).

However, this technique leaves the operating system somewhat confused. It continues trying to print the normal Mode 4 40-column text and 320-pixel-wide graphics. To use the system you will have to use the PRINT and PLOT commands very carefully!

The Video ULA

The Video ULA is accessed through two write-only locations. The Video Control Register at &FE20 controls a number of factors to do with the current mode. Complete details are given in *The Advanced User Guide*. As far as we are concerned it just sets which mode we are in.

Much more difficult to explain is the second register at &FE21 which gives us access to the palette. The high nibble of the location is the logical colour and the low nibble is the actual colour. By sending one byte to this location we can change one entry in the palette. So far, so good, but here comes the crunch. Just to make things more difficult, there are two complications. The first is that the actual colour nibble should contain the actual colour EXCLUSIVE-ORed with 7! The second is that to change all the colours in *any* mode you have to alter 16 entries in the palette.

For the 16-colour Mode 2 this is obvious. Each logical colour has one entry in the palette.

For the one-colour modes, entries 0 to 7 in the palette must all be set according to logical colour 0 and entries 8 to 15 must be set according to logical colour 1.

For four-colour modes, it is worse. To set each logical colour you need to set four entries as follows:

Logical colour	Entries to be changed
0	0 1 4 5
1	2 3 6 7
2	8 9 12 13
3	10 11 14 15

For example, in Mode 1, to do the equivalent of VDU19,2,3;0; we would have to do:

```

?&FE21=&84
?&FE21=&94
?&FE21=&C4
?&FE21=&D4

```

The reasons for all this are so involved that they are virtually unexplainable. However, I can assure you that there are very good reasons—I think! As far as we are concerned it is usually easier to use VDU19. However, we will see, later in this chapter, one example of where the added speed of the direct method is necessary. Notice that this method won't alter the copy of the palette that the operating system keeps, so that using OSWORD 11 to read the palette will give false answers.

Screen splitting

It would be very useful to be able to split the screen into two halves and have one half, say, in Mode 0 and the other half in Mode 1. This may seem impossible; but, using interrupts, it can be done! What we

need to do is to use the vertical sync interrupt to put the screen in Mode 0 and then set a timer to produce an interrupt, half-way down the screen, which we can use to put the screen in Mode 1. Because both are 20K modes it shouldn't be too difficult to change mode in the middle. The contents of the 6845 shouldn't need any changes, so all we need to change is the contents of the ULA.

Let's write two routines that will put the screen in Mode 1 from Mode 0 and Mode 0 from Mode 1 without clearing the screen. First of all, let us assume that the screen is already in Mode 0. To change to Mode 1, the first thing we need to do is change the video control register at &FE20 to set up the mode we are using. We also need to update the operating system's copy of this register at &248.

```
.mode1 LDA #&DB
        STA &FE20
        STA &248
```

(320-340)

Next we need to change the palette by writing to the register at &FE21. Colours 0 and 3 will already be correctly set so we only need to change 1 and 2. We need to change four bytes for each logical colour in Mode 1.

```
        LDX #7
.11     LDA m1,X
        STA &FE21
        DEX
        BPL 11
        RTS
.m1     EQU D &26366676
        EQU D &8494C4D4
```

(350-420)

We can do likewise for Mode 0.

```
.mode0 LDA #&9C
        STA &FE20
```

```

                STA &248
                LDX #7
.10            LDA m0,X
                STA &FE21
                DEX
                BPL 10
                RTS
.m0           EQU  &27376777
                EQU  &8090C000

```

(200-300)

Next we need to set up an interrupt routine in the interrupt vector (&204 and &205).

```

.init         SEI
                LDA &204
                STA &230
                LDA &205
                STA &231
                LDA #irq MOD256
                STA &204
                LDA #irq DIV256
                STA &205

```

(440-520)

We will also need to disable the centisecond clock interrupt and the analogue-to-digital converter interrupt, as these will tend to interfere with the program. If you need to leave these enabled, then you will have to put up with some flickering on the screen.

```

                LDA #&50
                STA &FE4E
                CLI
                RTS

```

(530-560)

The interrupt routine will first need to save the registers.

```
.irq    LDA &FC
        PHA
        TXA
        PHA
        TYA
        PHA
```

(580-630)

Next we need to check that the vertical sync caused the interrupt. If so, we need to switch to Mode 0.

```
LDA #2
BIT &FE4D
BEQ notsync
JSR mode0
```

(640-670)

We then need to set Timer 2 in the system VIA to count for half a screen and then produce an interrupt. So that we can move the boundary between the two modes easily, we put the boundary position in &70 and &71.

```
LDA &FE4B
AND #&DF
STA &FE4B
LDA &FE4E
ORA #&20
STA &FE4E
LDA &70
STA &FE48
LDA &71
STA &FE49
```

(680-770)

Finally we need to exit the interrupt routine by restoring the registers and jumping to the spare vector.

```
.exit   PLA
        TAY
        PLA
```

```
TAX
PLA
STA &FC
JMP (&230)
```

(780-840)

If the interrupt is not caused by the vertical sync then we need to check whether it is the timer interrupt. If so, we must clear the interrupt flag and switch to Mode 1.

```
.notsync LDA #&20
          BIT &FE40
          BEQ exit
          STA &FE40
          JSR mode1
          JMP exit
```

(850-900)

Now all that remains is to try an example.

```
10 *TV0,1
20 MODE0
30 MOVE0,512:DRAW1279,512
```

Notice that, because the change of modes itself takes some time, about one line of pixels on the screen will be garbled. To overcome this we must make this line white so that it is the same in both modes. This way, the transition should be invisible.

```
40 PROCass
50 !&70=9900
```

This sets up the position of the border. It may vary from machine to machine, so you may have to experiment to find the best value.

```
60 CALLinit
```

Finally we can draw a pattern on the screen as a demonstration.

```

70 FORA%=OT01:VDU29,A%*640;512;
80 FORB%=OT0511STEP16
90 MOVEB%*1.25,0:DRAW639,B%
100 DRAW639-B%*1.25,511:DRAW0,511-B%
110 DRAWB%*1.25,0:NEXT,
120 VDU26:FORA%=OT01279STEP2
130 IFRND(2)=2MOVEA%,0:DRAWA%,508
140 NEXT
150 GOTO150
160 DEFPROCass
170 FORpass%=OT02STEP2
180 P%=&A00
190 [OPTpass%
200 .mode0 LDA #&9C \ Change to
210 STA &FE20 \ Mode 0.
220 STA &248
230 LDX #7
240 .10 LDA m0,X
250 STA &FE21
260 DEX
270 BPL 10
280 RTS
290 .m0 EQU D &8090C0D0 \ Colour data
300 EQU D &27376777 \ for Mode 0.
310
320 .mode1 LDA #&D8 \ Change to
330 STA &FE20 \ Mode 1.
340 STA &248
350 LDX #7
360 .11 LDA m1,X
370 STA &FE21
380 DEX
390 BPL 11
400 RTS
410 .m1 EQU D &26366676 \ Colour data
420 EQU D &8494C4D4 \ for Mode 1.
430
440 .init SEI \ Initialise
450 LDA &204 \ interrupts.
460 STA &230
470 LDA &205
480 STA &231
490 LDA #irq MOD256
500 STA &204

```

```

510          LDA #irq DIV256
520          STA &205
530          LDA #&50
540          STA &FE4E
550          CLI
560          RTS
570
580 .irq     LDA &FC           \ Trap ints.
590          PHA
600          TXA
610          PHA
620          TYA
630          PHA
640          LDA #2           \ Check for
650          BIT &FE4D        \ v.sync.
660          BEQ notsync
670          JSR mode0        \ Change to
680          LDA &FE4B        \ Mode 0
690          AND #&DF         \ and start
700          STA &FE4B        \ counter for
710          LDA &FE4E        \ boundary.
720          ORA #&20
730          STA &FE4E
740          LDA &70
750          STA &FE48
760          LDA &71
770          STA &FE49
780 .exit   PLA
790          TAY
800          PLA
810          TAX
820          PLA
830          STA &FC
840          JMP (&230)
850 .notsync LDA #&20        \ Check for
860          BIT &FE4D        \ Timer.
870          BEQ exit
880          STA &FE4D
890          JSR mode1        \ Change to
900          JMP exit         \ Mode 1.
910 ]
920 NEXT
930 ENDPROC

```

Beware of using VDU19. You will also find that plotting lines in the bottom half of the screen has interesting effects.

Screen swapping

Some expensive computers nowadays have a system which allows for totally flicker-free animation. This system has two completely separate blocks of memory for graphics. While one image is being displayed the other, concealed, image is being redrawn. When the new image is complete the VDU switches cleanly between the two pages so that the new image is displayed, while the first is redrawn. By repeating this process the image need never be seen being redrawn.

By now you will have guessed that there is a method for doing this on the BBC Micro. Because two complete blocks of memory are needed, we can't use a 20K mode. For our purposes let's try and animate two Mode 4 screens. The two blocks of memory, each 10K long, will start at &3000 and &5800 respectively.

The first problem we need to look at is that before we can redraw on the concealed screen we must clear its 10K block. All we have to do is to set 10K of memory to zero. This sounds like a simple task using post-indexed indirect addressing. However, for clearing such a large amount of memory this addressing mode is too slow. We will need to sacrifice the short, neat but slow solution for the long but fast solution. By using absolute indexed addressing we can clear one 256-byte page very fast, like this:

```
.clear  LDA #0
        TAX
.loop   STA &3000,X
        INX
        BNE loop
        RTS
```

To clear 10K of memory we need to add an STA command for each page of memory we wish to clear—here, this means 40 STA commands. To save

typing each of these separately, we can use the power of the assembler. To start with, we need the first few commands. Notice that we disable interrupts first, to increase speed that extra little bit.

```
1000 DEFPROCass
1010 DIMmc%500
1020 FORpass%=0TO2STEP2
1030 P%=mc%
1040 [OPTpass%
1050 .lclear SEI
1060          LDA #0
1070          TAX
1080 .lloop
1090 ]
```

Next we set A% to a loop from &3000 to &5700 in steps of 256. Then we re-enter the assembler where we left off, reset the option, and set up the STA command with the variable A%. Then we exit the assembler again and end the loop.

```
1100 FORA%=&3000TO&5700STEP256
1110 [OPTpass%
1120          STA A%,X
1130 ]:NEXT
```

The result of all this is that we have assembled all 40 commands as required. This has shortened the assembly code considerably and shows the advantages of a powerful assembler! Finally we must end the machine code loop, and re-enable interrupts.

```
1140 [OPTpass%
1150          INX
1160          BNE lloop
1170          CLI
1180          RTS
```

We can write a similar routine for clearing the second 10K block.

```
1190 .hclear SEI
1200          LDA #0
```



```

1210          TAX
1220 .hloop
1230 ]
1240 FORA%=&5800T0&7F00STEP256
1250 [OPTpass%
1260          STA A%,X
1270 ]:NEXT
1280 [OPTpass%
1290          INX
1300          BNE hloop
1310          CLI
1320          RTS

```

Now we can write a routine which swaps the two screens around. The obvious way to do this is to swap the two sections of memory. However, this would be ridiculously slow. Instead, we will switch the start-of-screen RAM address register in the 6845 between the two blocks, so altering which block is being displayed.

We will need a variable as a flag that will tell us which of the two blocks is currently being displayed. We can use &70 for this—if it contains zero then the low block is currently being displayed; otherwise, it contains 255.

The first job the routine must do is to wait for the vertical sync so that the screens swap cleanly during the vertical sync period.

```

1330 .swap    LDA #19
1340          JSR &FFF4

```

Next we need to invert the contents of &70 by EORing it with 255. If it is now 255 then we want to swap to displaying the high block.

```

1350          LDA &70
1360          EOR #255
1370          STA &70
1380          BNE high

```

We now want to display the low block. To do this we must alter the start-of-display RAM register to point to the low block (we need only change the high byte

as the low byte is zero in both cases).

```
1390      LDA #12
1400      STA &FE00
1410      LDA #6
1420      STA &FE01
```

Next we need to ensure that any plotting or printing will be placed where it can't be seen in the high block. To do this we need to alter two bytes in the operating system work-space to the high byte of the address of the top left-hand corner of the screen on which we are plotting—here, &58. We also need to do a cursor-home to move all the cursors, etcetera. At this point all that remains is to clear the concealed screen ready for plotting.

```
1430      LDA #&58
1440      STA &34E
1450      STA &351
1460      LDA #30
1470      JSR &FFEE
1480      JMP hclear
```

Now we can use the same method to swap back again.

```
1490 .high LDA #12
1500      STA &FE00
1510      LDA #11
1520      STA &FE01
1530      LDA #&30
1540      STA &34E
1550      STA &351
1560      LDA #30
1570      JSR &FFEE
1580      JMP lclear
1590 ]
1600 NEXT
1610 ENDPROC
```

Now all that we need is an example of how to use the program.

```

10 MODE4:VDU23,1,0;0;0;0;
20 HIMEM=&3000
30 PROCass
40 CALLclear:?:&70=255:CALLswap

```

Notice that we need to set HIMEM to reserve 20K of screen memory, the lower 10K of which we need to clear. Note also that we need to call *swap* once just to get everything running smoothly—this saves having an initialisation routine.

```

50 FORA%=0TO1019STEP16:MOVEA%,0
60 DRAW1023,A%:DRAW1023-A%,1023
70 DRAW0,1023-A%:DRAWA%,0
80 CALLswap:NEXT
90 GOTO50

```

A BASIC swap

Of course, if you don't need to redraw the image completely each time, and hence don't need to clear the screen, you don't necessarily need to use machine code. So, just to show that good results don't always need machine code (though it helps), here is an analogue clock entirely in BASIC.

The program works, as before, in two Mode 4 screens. The first job is to write a procedure to draw a face without the hands.

```

1000 DEFPROCface
1010 GCOL0,1:P%=4
1020 FORA=0TOPI*2STEPPI/24
1030 MOVE0,0:PLOTTP%,512*SINA,512*COSA
1040 P%=85:NEXT
1050 GCOL0,0:P%=4
1060 FORA=0TOPI*2STEPPI/24
1070 MOVE0,0:PLOTTP%,492*SINA,492*COSA
1080 P%=85:NEXT

```

This draws the rim of the face. Next we need the dots for the hours and a dot at the centre. For speed it is better to use a defined character for this and position it with the VDU 5 'text at graphics cursor' mode.

```

1090 VDU23,224,0,&1C,&3E,&7F,&7F,&7F,
      &3E,&1C,5
1100 GCOLD,1:FORA=0TOPI*2STEPPI/6
1110 MOVE450*SINA-16,450*COXA+16:VDU224
1120 NEXT:MOVE-16,16
1130 VDU224,4,23,1,0;0;0;0;
1140 ENDPROC

```

Next we need a procedure that will draw the three hands in their correct positions. Notice that we have not specified the colour in this procedure, so it can be used both to draw the hands and remove them.

```

2000 DEFPROCchands(hours,minutes,seconds)
2010 A=seconds*PI/30:X=SINA:Y=COSA
2020 MOVEX*32,Y*32:DRAWX*420,Y*420
2030 A=minutes*PI/30:X=SINA:Y=COSA
2040 MOVEX*32+Y*5,Y*32-X*5
2050 MOVEX*32-Y*5,Y*32+X*5
2060 PLOT85,X*370+Y*5,Y*370-X*5
2070 PLOT85,X*370-Y*5,Y*370+X*5
2080 A=hours*PI/6:X=SINA:Y=COSA
2090 MOVEX*32+Y*12,Y*32-X*12
2100 MOVEX*32-Y*12,Y*32+X*12
2110 PLOT85,X*300+Y*12,Y*300-X*12
2120 PLOT85,X*300-Y*12,Y*300+X*12
2130 ENDPROC

```

Now we need the BASIC equivalent of the swap routine from the machine code program, but without the clear routines. We can use the variable S% instead of &70.

```

3000 DEFPROCswap
3010 *FX19
3020 S%=S%EOR1
3030 IFS%THEN?&FE00=12: ?&FE01=6:
      ?&34E=&58: ?&351=&58:VDU30:ENDPROC
3040 ?&FE00=12: ?&FE01=11: ?&34E=&30:
      ?&351=&30:VDU30:ENDPROC

```

Now we can write the main routine. The first job is to input the start time.

```
10 MODE7:INPUT"Hours, Minutes,  
Seconds",hours,minutes,seconds
```

Next we need to clear both screens. As we have no machine-code clearing routines the easiest way to do this is to go into Mode 0. Next we need to go into Mode 4 and set HIMEM to reserve the 20K of screen memory.

```
20 MODE0:MODE4:HIMEM=&3000
```

Now we must turn off the cursor and set the graphics origin to the middle of the screen. We must also set S% suitably.

```
30 S%=0:VDU23,1,0;0;0;0;29,640;512;
```

Next we need to draw the face on each of the two pages, having first called *swap* to set the paged graphics working.

```
40 PROCswap:PROCface:PROCswap:PROCface
```

Now we need to set T to the number of centiseconds that have elapsed since 12 o'clock. We also need to set *hours* so that it is the relevant distance between the hours. The minute and second hands will jump a minute and a second at a time, respectively.

```
50 T=seconds*100+minutes*6000+hours*360000  
60 hours=T/360000
```

Next we need to draw the hands in. We also need to keep a copy of where they are so that we can remove them later. We can then swap so that the face plus hands is visible and wait for the user to press a key to synchronise the clock.

```
70 GCOL0,1:PROChands(hours,minutes,seconds)  
80 s=seconds:m=minutes:h=hours  
90 PROCswap:A=GET
```

Now we must start the clock by setting TIME to T. For timing we will wait until TIME crosses the

hundred border. For this purpose we need to keep a copy of what TIME DIV100 equals at this point in t , so that when we are waiting to display the next second we can wait until TIME DIV100 doesn't equal t .

```
100 TIME=T
110 t=TIME DIV100
```

Now as we are displaying the time given by t we need to set up the time given by $t+1$ in the other block. To do this we must first remove the previous hands which are located at s , m and h . Then we must set these variables to the current hand positions ready for the next move.

```
120 GCOL0,0:PROChands(h,m,s)
130 s=seconds:m=minutes:h=hours
```

If TIME has passed 4320000 (12 hours in centiseconds) then we need to set it back 12 hours.

```
140 IFTIME>4319999 TIME=TIME-4320000
```

Next we must set up the new hands.

```
150 seconds=(t+1)MOD60
160 minutes=(t+1)DIV60 MOD60
170 hours=(t+1)/3600
```

However, the $+1$ for the hours makes practically no difference (it moves the hour hand on by 0.00027 of an hour) so we can ignore it. We are now ready to draw the new hands on the concealed screen.

```
170 hours=t/3600
180 GCOL0,1:PROChands(hours,minutes,seconds)
```

Now all we need to do is wait until the second is up and swap screens before repeating the whole process.

```
190 REPEATUNTILTIME DIV100<>t
200 PROCswap:GOTO110
```

One word of warning—because all printing is concealed, any error messages will never appear. If there is an error the machine will just appear to stop working. To get around this while you are typing in and debugging the program, try adding this line:

```
1 ONERRORMODE7:REPORT:PRINT" at line
";ERL:END
```

Three-dimensional graphics

Computers are being used increasingly now for producing three-dimensional graphics, particularly on television. In the home micro market there are an increasing number of three-dimensional games. In passing, it would be useful to take a look at some of the simple mathematics behind this subject.

We are faced with the problem of converting a set of three-dimensional coordinates into two-dimensional ones for plotting on the screen. We know from experience that an object further away appears smaller and we would probably guess (correctly, as it happens) that if an object goes twice as far away it looks half the size. Thus we could form a rule that apparent size is inversely proportional to distance.

Let's assume that we have three-dimensional axes X, Y and Z, where X and Y are horizontal and Z is vertical. Let's imagine that the origin is about 2 metres off the ground and that we place a camera at this point looking vertically downwards. Let's further assume that we have a square of cardboard. We place this horizontally at different levels with its centre always on the Z axis, photographing each position. Can we make a rule about where the corners will appear in each photograph? We can assume that the image on the photograph will still be a square. If we place a drawing horizontally under the camera and photograph it we would expect to get a precise copy of the original drawing. It might be bigger or smaller depending on what level we placed the drawing at.

From this we can guess that for one value of Z coordinate, given the X and Y coordinates of a point on that Z plane, the X and Y coordinates of the

point's image on the photograph, A and B, would be given by:

$$A = k X$$
$$B = k Y$$

We already know that the apparent size is inversely proportional to the distance from the camera to the point, so, as the camera is at the origin and we can take the Z coordinate of the plane as this distance, this suggests that A and B are given by:

$$A = k X/Z$$
$$B = k Y/Z$$

The constant k will be a scale factor—the power of lens we use, perhaps. As any point under the camera can be described as a point on a horizontal plane, we have here the equations for converting the 3D point into a 2D image.

If you didn't follow all that, don't worry. All you need remember is that by looking along the Z axis from the origin you can find the image of a point by using these equations.

As an example, let's try drawing a cube on the screen. The coordinates of the corners of this cube will be (10,10,40), (-10,10,40), (-10,-10,40), (10,-10,40), (10,10,60), (-10,10,60), (-10,-10,60) and (10,-10,60). First we need a procedure that will do the job of the PLOT command, but in 3D.

```
10000 DEFPROCplot(P%,X,Y,Z)
10010 PLOT P%,X*S/Z,Y*S/Z
10020 ENDPROC
```

Notice that the scale factor S will need to be defined at the beginning of the program. We can now draw our cube by drawing the nearest surface, then the four edges joining the front surface to the back surface, then the back surface itself.

```
10 MODE4:VDU23,1,0;0;0;0;29,640;512;
20 S=1000
30 PROCplot(4,10,10,40)
```



```

40 PROCplot(5,-10,10,40)
50 PROCplot(5,-10,-10,40)
60 PROCplot(5,10,-10,40)
70 PROCplot(5,10,10,40)
80 PROCplot(5,10,10,60)
90 PROCplot(4,-10,10,40)
100 PROCplot(5,-10,10,60)
110 PROCplot(4,-10,-10,40)
120 PROCplot(5,-10,-10,60)
130 PROCplot(4,10,-10,40)
140 PROCplot(5,10,-10,60)
150 PROCplot(5,-10,-10,60)
160 PROCplot(5,-10,10,60)
170 PROCplot(5,10,10,60)
180 PROCplot(5,10,-10,60)
190 END

```

This produces a recognisable cube but it is not the most exciting piece of art. It would be better if we could look at the cube from a different angle. However, our 3D equations will not let us do this. The solution, of course, is that if the mountain can't come to Mohammed then Mohammed must go to the mountain—we have to rotate the cube.

To do this we need to look at the method of rotating a 2D point about the origin. Those of you who have dabbled at all in matrices will know that the general rotation matrix about the origin, by an angle A , is:

$$\begin{pmatrix} \cos A & -\sin A \\ \sin A & \cos A \end{pmatrix}$$

For those of you who have not experienced the joys of matrices this will mean precisely nothing! But all you need to know is that if we have coordinates (X,Y) and we wish to rotate them about the origin by an angle A anti-clockwise, to new coordinates (P,Q) , then P and Q can be calculated using the two formulae:

$$P = X * \cos A - Y * \sin A$$

$$Q = X * \sin A + Y * \cos A$$

In three dimensions we can adapt these formulae by rotating about, say, the X axis. This means that the X coordinate of the point stays the same and we can use the two formulae above, except that we use Y and Z, to rotate the other two coordinates. So for our example program the PLOT routine becomes:

```
10000 DEFPROCplot(P%,X,Y,Z)
10010 Q=Y*COSA-Z*SINA
10020 R=Y*SINA+Z*COSA
10030 PLOTP%,X*S/R,Q*S/R
10040 ENDPROC
```

However, this rotates around our viewpoint. This doesn't help much. We really need to shift the coordinates so that the centre of the cube is over the origin, then rotate the cube about the X axis, and then shift the cube back again so that we can view it.

```
10000 DEFPROCplot(P%,X,Y,Z)
10010 Z=Z-DR
10020 Q=Y*COSA-Z*SINA
10030 R=Y*SINA+Z*COSA
10040 R=R+DR
10050 PLOTP%,X*S/R,Q*S/R
10060 ENDPROC
```

We also need to set the values of A and DR at line 20.

```
20 S=1000:A=-PI/5:DR=40
```

Let's now look at a more complicated example—drawing a cup or wine glass. This is not as difficult as it sounds, as a cup has rotational symmetry. This means that we need only store data for half of a slice through the axis of symmetry of the cup. We can then rotate this a number of times about the axis of symmetry and join all the adjacent points.

The first thing is our 3D procedure.

```
1000 DEFPROCplot(P%,X,Y,Z)
1010 P=Y*C-Z*I
1020 Q=Y*I+Z*C+D
```

```
1030 PLOTP%,X*S/Q,P*S/Q
1040 ENDPROC
```

Notice that, because the sine and cosine of the angle of rotation about the X axis remain constant as the angle remains constant, we can define these at the start of the program as I and C, and hence save a lot of calculation. Also, the initial coordinates will be centred on the origin so we can rotate these straight away. We then need to add a constant to the Z coordinate to shift the point away from the origin before we can view it.

The beginning of the program will need to look like this:

```
10 MODE1:VDU23,1,0;0;0;0;
20 VDU29,640;195;19,3,2;0;
30 D=300:S=2400
40 I=SINRAD240:C=COSRAD240
```

The first piece of data for the cross-section of the cup will need to be the number of points used to describe it. Because we will be joining the first point in this description to the second, we will need to keep two adjacent points in variables at one time. We can read in the number of points and the first point. The number of lines will be one less than the number of points. Each line will need a colour associated with it.

```
50 READN,R,Z:FORM=2TON
60 READR1,Z1,C%:GCOL0,C%
```

Before repeating the loop we will need to copy R1 and Z1 into R and Z. Now we are ready to rotate the line formed by these two points around the Z axis. At each step we must join the previous second point to the new second point and join the two new points together.

```
70 P%=4:FORA=0TOPI*2.01STEPPI/20
80 PROCp1ot(P%,R1*COXA,R1*SINA,Z1)
90 PROCp1ot(5,R*COXA,R*SINA,Z)
100 PROCp1ot(4,R1*COXA,R1*SINA,Z1)
```

```
110 P%=5:NEXT
120 R=R1:Z=Z1
130 NEXT
140 END
```

Finally we need the data.

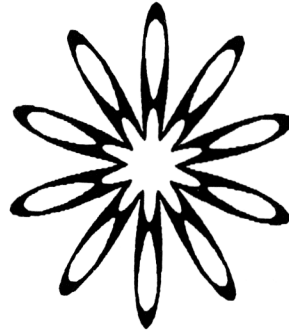
```
150DATA7,0,-5,32,-5,3,32,0,2,8,10,2,8,
35,1,24,40,1,24,85,3
```

This is, of course, only a simple example of 3D graphics.

FILL ROUTINES

Although the operating system contains a very good selection of graphics routines it lacks several useful sets of commands such as a sprite routine. Another thing the operating system lacks is an efficient method of filling shapes with colour. In fact the only command the operating system provides for producing blocks of colour is the PLOT85 triangle fill routine. This command can be used to very good affect for producing solid circles, etcetera, by approximating the shape in question by a series of overlapping triangles. A time comes, however, when there is a need for a routine that will fill all the area within a boundary. Take the following shape for example:

8.1 A sample shape



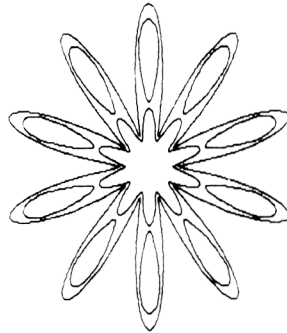
This could be drawn using triangles but that would be very complicated. What we want to be able to do is to draw the outline and then call a machine code routine, first specifying a starting point inside the outline, that will fill the shape for us. The first thing we need is a program that will produce the outline of the shape we want to fill.

```

10 REM Program to draw outline.
20 MODE0
30 VDU23,1,0;0;0;0;
40 P%=4
50 FORA=0TOPI*2STEPPI/100
60 R%=COS(A*10)*200+300
70 PLOTP%,640+SIN(A)*R%,512+COS(A)*R%
80 P%=5
90 NEXT
100 P%=4
110 FORA=0TOPI*2STEPPI/80
120 R%=COS(A*10)*50+130
130 PLOTP%,640+SIN(A)*R%,512+COS(A)*R%
140 P%=5
150 NEXT
160 FORA=0TOPI*2STEPPI/5
170 P%=4
180 FORB=0TOPI*2STEPPI/30
190 C=A+SIN(B)*PI/32
200 R%=325+125*COS(B)
210 PLOTP%,SIN(C)*R%+640,COS(C)*R%+512
220 P%=5
230 NEXT,

```

8.2 A sample outline



A BASIC fill

Next we need a fill routine. Instead of writing the machine code program straightaway we will write an experimental BASIC one first. This is often a good idea with complicated machine code programs as it is easier to debug a BASIC program than a machine code one. Once the BASIC program

works it is relatively easy to convert it to machine code. For our BASIC fill program we will use a procedure. The parameters we will need are the X and Y coordinates of a point within the outline which we've already drawn, and a definition of what the routine is to consider as the outline. The easiest way to do this is to specify a colour and say that any point of a different colour is part of the outline. The colour we specify, then, is the background colour of the shape. Here we can use the X and Y coordinates 640 and 32 for the point within the outline; the background colour is zero. So our example will need the lines:

```
500 PROCfill(640,32,0)
510 END
```

We are going to use X% and Y% for the coordinates of the point we are looking at any one time—the fill 'cursor'. For convenience, then, we can define procedures for moving this cursor up, down, left and right one pixel at a time, which is necessary as we are exploring the area just around the current point. So that we can use these procedures in different graphics modes we need to specify a variable M%, which will differ for different modes, which is the amount by which we have to alter X% to move one pixel left or right. This variable should be defined at the beginning of the program (line 20) when we change mode. We replace the old line 20 by:

```
20 MODE0:M%=2
```

Our up, down, left and right procedures are:

```
2000 DEFPROCup:Y%=Y%+4:ENDPROC
2010 DEFPROCdown:Y%=Y%-4:ENDPROC
2020 DEFPROCleft:X%=X%-M%:ENDPROC
2030 DEFPROCright:X%=X%+M%:ENDPROC
```

We must also define a function that will return 'true' only if X% and Y% point to a pixel whose colour is the background colour (C%).

```
2040 DEFFNbackground=(POINT(X%,Y%)=C%)
```

Here, (POINT(X%,Y%)=C%) is true only if both sides of the = sign agree.

Now we can start to try and write the fill procedure itself. For this purpose let's take a simple example shape which we can use to work out the program. The cross represents the cursor position (though in the real program this will be invisible). The cursor is initially positioned at the point we have used to define the inside of the outline.

See diagram 8.3 on page 153.

We are going to fill the shape with a series of horizontal lines, each one pixel thick. We will start from the bottom of the outline and work up. For this reason, we must first find a bottom point to start at. Imagine the outline as holding water: we need a place where water will collect—a hollow of some kind.

The first thing we can do is go vertically down until we hit the boundary.

```
1000 DEFPROCfill(X%,Y%,C%)
1010 PROCdown:IFFNbackground THEN1010
      ELSE PROCup
```

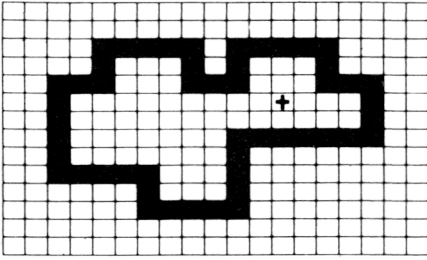
This moves the cursor down until the boundary is reached and then moves it up one pixel so that the cursor is on the bottom background pixel.

See diagram 8.4 on page 153.

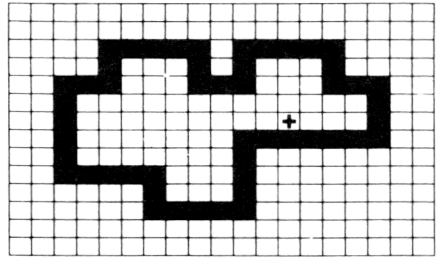
Next we can go left in a horizontal line until we hit the boundary. We then need to move right and, for reasons we shall see later, we need to make a copy of these coordinates in X1% and Y1%.

```
1020 PROCleft:IFFNbackground THEN1020
1030 PROCright:X1%=X%:Y1%=Y%
```

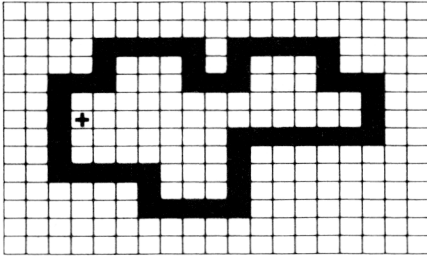
However, at this point (diagram 8.5, page 153) we can still go down. We need to check whether there



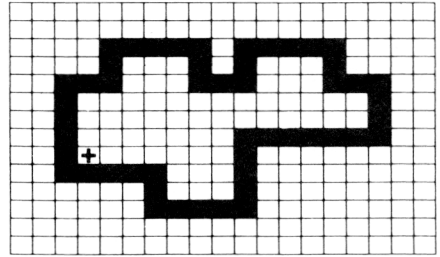
8.3



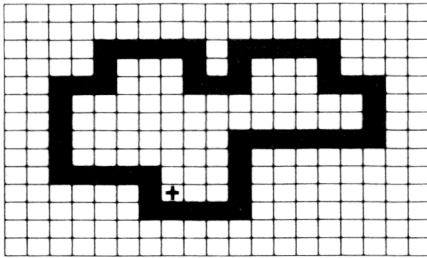
8.4



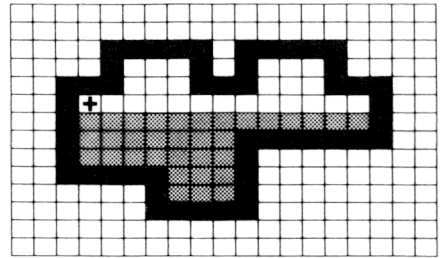
8.5



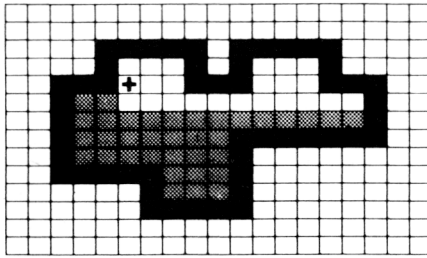
8.6



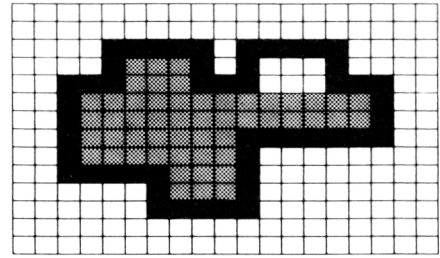
8.7



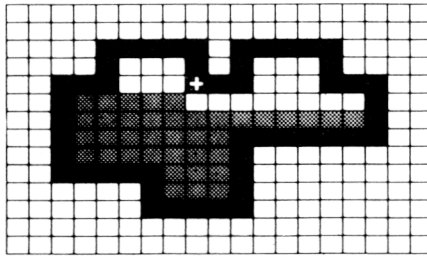
8.8



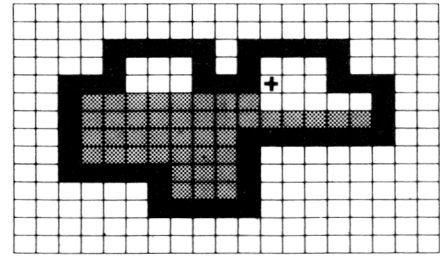
8.9



8.10



8.11



8.12

is still background below the cursor, and, if so, go back to line 1010 to find the bottom of the section.

```
1040 PROCdown:IFFNbackground THEN1010
```

By the time we get back to line 1020 again we have reached a new position.

See diagram 8.6 on page 153.

At line 1020 we cannot go further left. At line 1040 there is no background below. We now have to go right in a horizontal line until we reach the boundary on the other side. As we go, we can check below for any background. If this is found then we go back to line 1010 yet again to find the bottom.

```
1050 PROCup:PROCrigh:IFFNbackground THEN1040
```

By the time we get back to this point again we have found the bottom of the shape. The coordinates of the leftmost point of the bottom line are stored in X1% and Y1%. We must place the cursor back at this position.

```
1060 X%=X1%:Y%=Y1%
```

See diagram 8.7 on page 153.

The obvious thing to do now is to fill this bottom line. However, as we do this we must check to see if we have finished filling the outline. As we fill each pixel of this line we must check the pixel above to see if it is background. If no background is found above then we have finished filling the shape. If some background is found then we must make a note in X1% and Y1% of the first pixel of background we find and set a flag to tell us that there is more to be filled above. What we will do is use the variable F% as a flag. At the start of filling the line, we set this to 0.

```
1060 X%=X1%:Y%=Y1%:F%=0
```

Then, when the first background pixel is found

above, we set F% to 1 and store the coordinates in X1% and Y1%. Any further background points we find, F% will already be set to 1 and we will know not to alter X1% and Y1%. Then when the line is filled, if F% is still zero, no background has been found on the next line up and we have finished. Otherwise, we can move the cursor to the coordinates in X1% and Y1% and start to fill the next line.

So the first job to do, before we fill each pixel on the current line, is to move up and check whether there is background above.

```
1070 PROCup:IFNOTFbackground THEN1090
```

If background has been found and F% is still 0, then we must set F% to 1 and copy the cursor coordinates into X1% and Y1%.

```
1080 IFF%=0 F%=1:X1%=X%:Y1%=Y%
```

Next we can fill the pixel on the line we are currently working on.

```
1090 PROCdown:PLOT69,X%,Y%
```

Next we move right a pixel and, unless the boundary has been reached, go back and fill the next pixel.

```
1100 PROCright:IFFbackground THEN1070
```

If F% is now 1 then we can go back and fill the next line up. We need to go back to line 1020 for this, to find the left-hand end of the line.

```
1110 IFF%=1 X%=X1%:Y%=Y1%:GOTO1020
```

Otherwise, we have finished and can return from the procedure.

```
1120 ENDPROC
```

This would appear to be a perfectly good fill

routine. However, it will only work until it reaches this point:

See diagram 8.8 on page 153.

At line 1080 onwards, it will have reached this point:

See diagram 8.9 on page 153.

It will then keep a copy of this point in X1% and Y1% and set F% to 1. It will then ignore any other background that it finds on the line above. Thus when it fills the next line up, it will only fill the left-hand area of the shape. The final result, when the computer thinks it has finished, will be this:

See diagram 8.10 on page 153.

To solve this problem we need to re-write lines 1070 to 1120 leaving gaps for the lines we need to add.

```
1070 PROCup:IFFNbackground THEN1100
1090 GOTO1120
1100 IFF%=0 F%=1:X1%=X%:Y1%=Y%
1120 PROCdown:PLOT69,X%,Y%
1130 PROCright:IFFNbackground THEN1070
```

See diagram 8.11 on page 153.

When the cursor reaches this point we need to set F% to 2. This shows that we have found the entire width of the first area. Thus if boundary is detected and F% is already 1, then we need to set F% to 2.

```
1080 IFF%=1 F%=2
```

See diagram 8.12 on page 153.

When the cursor reaches this point, F% will be 2. Thus we know that the program must make a decision about which area to fill first. We will say that the program will keep a note of the position of the first

area (already stored in X1% and Y1%) and carry on filling from where the cursor is, as if it is starting a new horizontal line.

```
1110 IFF%=2THEN1170
```

However, in filling the second area, the program might come across another decision point and have to keep a note of a second set of coordinates.

What we need is a buffer in which we can place a whole list of coordinates. We can use an array for this. We are unlikely to need to keep a note of more than 64 sets of coordinates so we can dimension an array 64 by 2. Because we can use entry zero in an array, the following command will be sufficient.

```
5 DIM S%(63,1)
```

We can then say that the first point saved on this buffer will have X and Y coordinates stored in S%(0,0) and S%(0,1), and so on.

We also need a pointer to keep track of how many points we have kept a note of at any time. For this we can use the variable P%. This needs to be set to zero on entry to the procedure.

```
1000 DEFPROCfill(X%,Y%,C%):P%=0
```

When we reach a decision point, we must place the coordinates of the first area (currently in X1% and Y1%) in the first clear entry of the buffer and add one to the pointer P%. Then we can move down a pixel, back to the line we were originally filling. We need to set F% to 0, so that the program carries on looking for areas above the current line that need filling, and then go back to line 1070.

```
1170 S%(P%,0)=X1%:S%(P%,1)=Y1%:P%=P%+1
```

```
1180 PROCdown:F%=0:GOTO1070
```

If the program does not reach a decision point then it will finish filling the line it is on. At this stage, if there is more to be filled on the next line up, F% will be either 1 or 2 and the coordinates of the area to be

filled will be in X1% and Y1%.

```
1140 IFF%>0 X%=X1%:Y%=Y1%:GOTO1020
```

If not then the program has reached a dead-end. However, there may still be some points stored in the buffer that need to be explored. If P% is 0 then we have finished the entire job and can return from the procedure.

```
1150 IFP%=0 ENDPROC
```

If P% is not 0, then we need to subtract one from the pointer, P%; remove the most recent point from the stack; and start filling from this point.

```
1160 P%=P%-1:X%=S%(P%,0):Y%=S%(P%,1):GOTO1020
```

This completes the fill program. To make it easier to type in, here is the complete program.

```
5 DIM S%(63,1)
10 REM Program to draw outline.
20 MODE0:M%=2
30 VDU23,1,0;0;0;0;
40 P%=4
50 FORA=0TOPI*2STEPPI/100
60 R%=COS(A*10)*200+300
70 PLOTP%,640+SIN(A)*R%,512+COS(A)*R%
80 P%=5
90 NEXT
100 P%=4
110 FORA=0TOPI*2STEPPI/80
120 R%=COS(A*10)*50+130
130 PLOTP%,640+SIN(A)*R%,512+COS(A)*R%
140 P%=5
150 NEXT
160 FORA=0TOPI*2STEPPI/5
170 P%=4
180 FORB=0TOPI*2STEPPI/30
190 C=A+SIN(B)*PI/32
200 R%=325+125*COS(B)
210 PLOTP%,SIN(C)*R%+640,COS(C)*R%+512
220 P%=5
```

```

230 NEXT,
500 PROCfill(640,32,0)
510 END
1000 DEFPROCfill(X%,Y%,C%):P%=0
1010 PROCdown:IFFNbackground THEN1010
      ELSE PROCup
1020 PROCleft:IFFNbackground THEN1020
1030 PROCright:X1%=X%:Y1%=Y%
1040 PROCdown:IFFNbackground THEN1010
1050 PROCup:PROCright:IFFNbackground THEN1040
1060 X%=X1%:Y%=Y1%:F%=0
1070 PROCup:IFFNbackground THEN1100
1080 IFF%=1 F%=2
1090 GOTO1120
1100 IFF%=0 F%=1:X1%=X%:Y1%=Y%
1110 IFF%=2THEN1170
1120 PROCdown:PLOT69,X%,Y%
1130 PROCright:IFFNbackground THEN1070
1140 IFF%>0 X%=X1%:Y%=Y1%:GOTO1020
1150 IFF%=0 ENDPROC
1160 P%=P%-1:X%=S%(P%,0):Y%=S%(P%,1):GOTO1020
1170 S%(P%,0)=X1%:S%(P%,1)=Y1%:P%=P%+1
1180 PROCdown:F%=0:GOTO1070
2000 DEFPROCup:Y%=Y%+4:ENDPROC
2010 DEFPROCdown:Y%=Y%-4:ENDPROC
2020 DEFPROCleft:X%=X%-M%:ENDPROC
2030 DEFPROCright:X%=X%+M%:ENDPROC
2040 DEFFNbackground=(POINT(X%,Y%)=C%)

```

A machine code fill

If you run the BASIC program you will find that it takes about eight minutes to fill our outline. This is clearly impracticable for serious use so we need to re-write the routine in machine code. The quickest way to do this is to simply convert the program, line for line, into machine code.

First we must define the zero page locations we will use to replace each of the BASIC variables.

&70 and &71	X% low and high
&72 and &73	Y% low and high
&74	colour of current pixel
&75	C%—background colour
&76	P%—stack pointer

&77	F%—decision flag
&78 and &79	X1% low and high
&7A and &7B	Y1% low and high

We need a procedure to assemble the machine code.

```

1000 DEFPROCass
1010 DIMmc%350
1020 FORpass%=0TO2STEP2
1030 P%=mc%
1040 [OPTpass%

```

Now we can start writing the routine. We can improve the speed by disabling interrupts during the fill. Then we can look at the first line of the BASIC which defined the procedure and set P% to zero. This becomes:

```

1050 .fill SEI
1060 LDA #0
1070 STA &76

```

Where we used procedures in the BASIC example for left, right, up and down, we can use subroutines in machine code with the same labels. For the function *background* we can write a subroutine called *backgr* which sets the zero flag only if the pixel under the cursor is of the background colour.

The next line of the BASIC was:

```

1010 PROCdown:IFFNbackground THEN1010
ELSE PROCup

```

This, then, becomes the following piece of machine code.

```

1080 .loop1 JSR down
1090 JSR backgr
1100 BEQ loop1
1110 JSR up

```

Next we had the lines:

```
1020 PROCleft:IFFNbackground THEN1020
1030 PROCright:X1%=X%:Y1%=Y%
```

This becomes:

```
1120 .loop2 JSR left
1130 JSR backgr
1140 BEQ loop2
1150 JSR right
1160 LDA &70
1170 STA &78
1180 LDA &71
1190 STA &79
1200 LDA &72
1210 STA &7A
1220 LDA &73
1230 STA &7B
```

Then the lines:

```
1040 PROCdown:IFFNbackground THEN1010
1050 PROCup:PROCright:IFFNbackground THEN1040
1060 X%=X1%:Y%=Y1%:F%=0
```

This becomes:

```
1240 .loop3 JSR down
1250 JSR backgr
1260 BEQ loop1
1270 JSR up
1280 JSR right
1290 JSR backgr
1300 BEQ loop3
1310 LDA &78
1320 STA &70
1330 LDA &79
1340 STA &71
1350 LDA &7A
1360 STA &72
1370 LDA &7B
1380 STA &73
1390 .loop4 LDA #0
1400 STA &77
```

Note the label *loop4*. By jumping to here when we reach the equivalent of line 1180 of the BASIC, we can save two commands.

Next we had:

```
1070 PROCup:IFFNbackground THEN1100
1080 IFF%=1 F%=2
1090 GOTO1120
1100 IFF%=0 F%=1:X1%=X%:Y1%=Y%
1110 IFF%=2THEN1170
```

This becomes:

```
1410 .loop5 JSR up
1420 JSR backgr
1430 BEQ skip1
1440 LDA &77
1450 CMP #1
1460 BNE point
1470 LDA #2
1480 STA &77
1490 BNE point
1500 .skip1 LDA &77
1510 BNE skip2
1520 LDA #1
1530 STA &77
1540 LDA &70
1550 STA &78
1560 LDA &71
1570 STA &79
1580 LDA &72
1590 STA &7A
1600 LDA &73
1610 STA &7B
1620 .skip2 LDA &77
1630 CMP #2
1640 BEQ skip5
```

We then had the line:

```
1120 PROCdown:PLOT69,X%,Y%
```

This codes into machine code using the VDU25 equivalent of the PLOT command.

```
1650 .point JSR down
1660 LDA #25
1670 JSR &FFEE
1680 LDA #69
1690 JSR &FFEE
1700 LDA &70
1710 JSR &FFEE
1720 LDA &71
1730 JSR &FFEE
1740 LDA &72
1750 JSR &FFEE
1760 LDA &73
1770 JSR &FFEE
```

```
1130 PROCright:IFFNbackground THEN1070
```

Becomes:

```
1780 JSR right
1790 JSR backgr
1800 BEQ loop5
```

Then we had the lines:

```
1140 IFF%>0 X%=X1%:Y%=Y1%:GOTO1020
1150 IFP%=0 ENDPROC
```

This becomes:

```
1810 LDA &77
1820 BEQ skip3
1830 LDA &78
1840 STA &70
1850 LDA &79
1860 STA &71
1870 LDA &7A
1880 STA &72
1890 LDA &7B
1900 STA &73
1910 JMP loop2
1920 .skip3 LDA &76
1930 BNE skip4
1940 CLI
1950 RTS
```

Next we have the first mention of the buffer. For this we can use the stack. The points can be pushed on the stack, four bytes each, and pulled off again in the reverse order. Location &76 can be used to keep track of how many points are on the stack, as before.

The next line from the BASIC was:

```
1160 P%=P%-1:X%=S%(P%,0):Y%=S%(P%,1):GOTO1020
```

This, then, becomes:

```
1960 .skip4   DEC  &76
1970         PLA
1980         STA  &73
1990         PLA
2000         STA  &72
2010         PLA
2020         STA  &71
2030         PLA
2040         STA  &70
2050         JMP  loop2
```

The next two lines from the BASIC were:

```
1170 S%(P%,0)=X1%:S%(P%,1)=Y1%:P%=P%+1
1180 PROCdown:F%=0:GOTO1070
```

These become:

```
2060 .skip5   LDA  &78
2070         PHA
2080         LDA  &79
2090         PHA
2100         LDA  &7A
2110         PHA
2120         LDA  &7B
2130         PHA
2140         INC  &76
2150         JSR  down
2160         JMP  loop4
```

Finally we have the five subroutines. The four move routines code rather obviously.

```
2000 DEFPROCup:Y%=Y%+4:ENDPROC
2010 DEFPROCdown:Y%=Y%-4:ENDPROC
2020 DEFPROCleft:X%=X%-M%:ENDPROC
2030 DEFPROCright:X%=X%+M%:ENDPROC
```

These become:

```
2170 .up      LDA &72
2180          CLC
2190          ADC #4
2200          STA &72
2210          BCC uskip
2220          INC &73
2230 .uskip   RTS

2240 .down    LDA &72
2250          SEC
2260          SBC #4
2270          STA &72
2280          BCS dskip
2290          DEC &73
2300 .dskip   RTS

2310 .left    LDA &70
2320          SEC
2330          SBC #M%
2340          STA &70
2350          BCS lskip
2360          DEC &71
2370 .lskip   RTS

2380 .right   LDA &70
2390          CLC
2400          ADC #M%
2410          STA &70
2420          BCC rskip
2430          INC &71
2440 .rskip   RTS
```

For the last routine, we need to use OSWORD 9 (the machine code equivalent of POINT). For this, a parameter block must be set-up with the X coordinate in the first two bytes (low then high) and the Y coordinate in the next two bytes (low then high).

The OSWORD routine then places the colour of the pixel in the fifth byte of the parameter block.

We already have the X coordinate in &70 and &71, and the Y coordinate in &72 and &73; we also want the colour returned in &74. So, all we need to do is set the X and Y registers (low, high) to point to location &70, set the accumulator to 9 and call OSWORD. We can then compare the colour returned in &74 with the background colour in &75. If they are the same then the zero flag will be set as required.

```
2450 .backgr LDA #9
2460         LDX #&70
2470         LDY #0
2480         JSR &FFF1
2490         LDA &74
2500         CMP &75
2510         RTS
2520 ]
2530 NEXT
2540 ENDPROC
```

This finishes the machine code but we still need a BASIC procedure that calls the routine.

```
950 DEFPROCfill(X%,Y%,C%)
960 !&70=X%+Y%*&10000
970 ?&75=C%
980 CALLfill
990 ENDPROC
```

As an example try adding the BASIC example section (lines 10 to 510) from the previous program. We don't now need line 5 as we are using the machine stack; but we need to call the assembly procedure once M% has been set.

```
25 PROCass
```

If you run this you will find that it is about eight times faster than the BASIC version. This routine will work in any mode so long as M% is set correctly before the assembler procedure is called.

A faster fill

For most purposes this routine is fast enough. However, there will be occasions when a still faster routine is needed. The things which slow down this routine are the operating system calls. The routines we have used are all written so as to be as flexible as possible. However, this slows them down considerably. If we are prepared to limit the fill routine to working in only one mode then we can write our own versions of the operating system routines which will be much faster.

To do this we need to use a different method of storing the coordinates of a point.

As an example of what is possible we can write a Mode 0 fill routine that will fill our example shape. As Mode 0 is a two-colour mode this shouldn't be too difficult. To simplify the program further still, we will assume that the background colour is zero.

Each pixel in Mode 0 is represented by one bit in the memory. So, instead of X% and Y% from our BASIC example, we need a different method of describing which pixel we are looking at. The best way to do this is to use &70 and &71 to store the address of the byte in the memory; and another byte, at &72, to describe which bit of that byte we are interested in. This byte will contain a one in the bit corresponding to the bit we are interested in and zeros in all the other bits. The reason for this is that we can then load the byte from the memory into the accumulator and AND it with the contents of &72 to leave either zero if the pixel was black or not zero if the pixel was white. This technique is sometimes called 'bit masking'.

For convenience and speed we will also keep the character column position (that is, the number of pixels along divided by eight) in &73. This will make it easier to check whether we have moved off the edge of the screen.

The first change we need to make to our machine code program is to add a routine at the beginning to convert the X and Y coordinates into an address, a bit mask, and a column number. On entry to the routine we must specify that &78 and &79 contain the X coordinate, and &7A and &7B contain the Y coordinate, in pixels, where the top left-hand corner

of the screen is (0,0). This means that the BASIC procedure for calling the fill routine now becomes:

```
960 DEFPROCfill(X%,Y%)
970 !&78=X%DIV2+&10000*(255-Y%DIV4)
980 CALLfill
990 ENDPROC
```

The first job for the machine code (after disabling interrupts and setting P% to zero) is to work out the address of the byte. This will be:

$\&3000 + 640 * Y \% DIV 8 + X \% AND \&FFF8 + Y \% MOD 8$.

So the first section of the routine looks like this:

```
.fill    SEI            \ Disable interrupts.
         LDA #0         \ Set P% to 0
         STA &76        \
         LDA &7A        \ Find
         AND #&F8       \ Y% DIV 8
         LSR A          \ times 2
         LSR A          \ for look 0.S.
         TAX           \ ROM table.
         LDA &7A        \ Y% MOD 8
         AND #7         \
         CLC           \
         ADC &C376,X    \ look up 640
         STA &70        \ times table
         LDA &C375,X    \
         CLC           \
         ADC #&30       \ add &3000
         STA &71        \
         LDA &78        \ add X% AND
         AND #&F8       \ &FFF8
         CLC           \
         ADC &70        \
         STA &70        \
         LDA &71        \
         ADC &79        \
         STA &71        \
```

(1050-1290)

If you are not sure how this works, look at chapter 10 on sprites, as this routine uses a similar section.

Next we must calculate the 'bit mask'. This must have the value $\&80$ if $X\%AND7$ is zero and $\&01$ if it is seven. To calculate it we can place the value $\&80$ in location $\&72$ and then shift it right the number of times specified by $X\%AND7$.

```
                LDA #&80
                STA &72
                LDA &78
                AND #7
                TAX
                BEQ skipA
.loopA          LSR &72
                DEX
                BNE loopA
.skipA         ...
```

(1300-1380)

Lastly we need to calculate the column number. This will be the X coordinate divided by eight. As the X coordinate cannot be larger than 639 we can divide it by two, twice, to get a number that cannot be larger than 159. Then we only need to shift the low byte for the last division by two. This saves one command.

```
.skipA         LSR &79
                ROR &78
                LSR &79
                ROR &78
                LSR &78
                LDA &78
                STA &73
```

(1390-1450)

From here on the program is similar to the previous machine code version. As the X and Y coordinates are still stored as four bytes from $\&70$ to $\&73$ the commands $X1\% = X\%:Y1\% = Y\%$ etcetera, will still be the same. The next thing we need to change is where a point is actually plotted. As we have assumed that the background colour is zero, the fore-

ground colour must be one. This means that to plot a point we need to set the relevant bit in the memory to one. To do this we can simply load the byte into the memory, OR it with the bit mask and store it back in the memory. So the code after the label *point* becomes:

```
.point JSR down
        LDY #0
        LDA (&70),Y
        ORA &72
        STA (&70),Y
        JSR right
        ...
```

Finally we need to rewrite the routines *up*, *down*, *left*, *right* and *backgr*. Here, *up* must first check if the point is off the top of the screen. If so, then it must exit the routine. If not, then it must check the least significant three bits of the address. If these are zero then we need to go up a whole character line. Otherwise we need to take one away from the address.

```
.up     LDA &71
        CMP #&30
        BCC up2
        LDA &70
        AND #7
        BNE up1
        LDA &70
        SEC
        SBC #&79
        STA &70
        LDA &71
        SBC #2
        STA &71
        RTS
.up1    DEC &70
.up2    RTS
```

(2480-2630)

down is similar.

```

.down    LDA &71
         BMI down2
         LDA &70
         AND #7
         CMP #7
         BNE down1
         LDA &70
         CLC
         ADC #&79
         STA &70
         LDA &71
         ADC #2
         STA &71
         RTS
.down1   INC &70
.down2   RTS

```

(2650-2800)

left needs to check that the pixel is on the screen and, if so, shift the bit mask left a bit; if the carry is then set, it needs to subtract eight from the address, decrement the column number and set the bit mask to one; *right* is similar.

```

.left    LDA &73
         BMI left1
         ASL &72
         BCC left1
         LDA #1
         STA &72
         DEC &73
         LDA &70
         SEC
         SBC #8
         STA &70
         LDA &71
         SBC #0
         STA &71
.left1   RTS
.right   LDA &73
         CMP #80
         BEQ right1

```

```

LSR &72
BCC right1
LDA #128
STA &72
INC &73
LDA &70
CLC
ADC #8
STA &70
LDA &71
ADC #0
STA &71
.right1 RTS

```

(2820-3130)

Lastly we need to deal with *backgr*. This must first check that the point is on the screen. If so, then it can load the byte from the memory, AND it with the bit mask and return to the main program with zero in the accumulator only if the pixel is background.

```

.backgr LDA &73
        BMI off
        CMP #80
        BEQ off
        LDA &71
        BMI off
        CMP #&30
        BCC off
        LDY #0
        LDA (&70),Y
        AND &72
        BNE off
        LDA #0
        RTS
.off   LDA #1
        RTS

```

(3150-3300)

We now have an effective routine. Here is a complete listing of it:

```

SPROCass
10REM Program to draw outline.
20MODE0
30VDU23,1,0;0;0;0;
40P%=4
50FORA=0TOPI*2STEPPI/100
60R%=COS(A*10)*200+300
70PLOTP%,640+SIN(A)*R%,512+COS(A)*R%
80P%=5
90NEXT
100P%=4
110FORA=0TOPI*2STEPPI/80
120R%=COS(A*10)*50+130
130PLOTP%,640+SIN(A)*R%,512+COS(A)*R%
140P%=5
150NEXT
160FORA=0TOPI*2STEPPI/5
170P%=4
180FORB=0TOPI*2STEPPI/30
190C=A+SIN(B)*PI/32
200R%=325+125*COS(B)
210PLOTP%,SIN(C)*R%+640,COS(C)*R%+512
220P%=5
230NEXT,
500PROCfill(640,32)
510END
960DEFPROCfill(X%,Y%)
970!&78=X%DIV2+&10000*(255-Y%DIV4)
980CALLfill
990ENDPROC
1000DEFPROCass
1010DIMmc%450
1020FORpass%=0TO2STEP2
1030P%=mc%
1040[OPTpass%
1050.fill SEI
1060 LDA #0
1070 STA &76
1080 LDA &7A
1090 AND #&F8
1100 LSR A
1110 LSR A
1120 TAX
1130 LDA &7A

```

1140	AND #7
1150	CLC
1160	ADC &C376,X
1170	STA &70
1180	LDA &C375,X
1190	CLC
1200	ADC #&30
1210	STA &71
1220	LDA &78
1230	AND #&F8
1240	CLC
1250	ADC &70
1260	STA &70
1270	LDA &71
1280	ADC &79
1290	STA &71
1300	LDA #&80
1310	STA &72
1320	LDA &78
1330	AND #7
1340	TAX
1350	BEQ skipA
1360.loopA	LSR &72
1370	DEX
1380	BNE loopA
1390.skipA	LSR &79
1400	ROR &78
1410	LSR &79
1420	ROR &78
1430	LSR &78
1440	LDA &78
1450	STA &73
1460.loop1	JSR down
1470	JSR backgr
1480	BEQ loop1
1490	JSR up
1500.loop2	JSR left
1510	JSR backgr
1520	BEQ loop2
1530	JSR right
1540	LDA &70
1550	STA &78
1560	LDA &71
1570	STA &79

1580	LDA &72
1590	STA &7A
1600	LDA &73
1610	STA &7B
1620.loop3	JSR down
1630	JSR backgr
1640	BEQ loop1
1650	JSR up
1660	JSR right
1670	JSR backgr
1680	BEQ loop3
1690	LDA &78
1700	STA &70
1710	LDA &79
1720	STA &71
1730	LDA &7A
1740	STA &72
1750	LDA &7B
1760	STA &73
1770.loop4	LDA #0
1780	STA &77
1790.loop5	JSR up
1800	JSR backgr
1810	BEQ skip1
1820	LDA &77
1830	CMP #1
1840	BNE point
1850	LDA #2
1860	STA &77
1870	BNE point
1880.skip1	LDA &77
1890	BNE skip2
1900	LDA #1
1910	STA &77
1920	LDA &70
1930	STA &78
1940	LDA &71
1950	STA &79
1960	LDA &72
1970	STA &7A
1980	LDA &73
1990	STA &7B
2000.skip2	LDA &77
2010	CMP #2

2020	BEQ skip5
2030.point	JSR down
2040	LDY #0
2050	LDA (&70),Y
2060	ORA &72
2070	STA (&70),Y
2080	JSR right
2090	JSR backgr
2100	BEQ loop5
2110	LDA &77
2120	BEQ skip3
2130	LDA &78
2140	STA &70
2150	LDA &79
2160	STA &71
2170	LDA &7A
2180	STA &72
2190	LDA &7B
2200	STA &73
2210	JMP loop2
2220.skip3	LDA &76
2230	BNE skip4
2240	CLI
2250	RTS
2260.skip4	DEC &76
2270	PLA
2280	STA &73
2290	PLA
2300	STA &72
2310	PLA
2320	STA &71
2330	PLA
2340	STA &70
2350	JMP loop2
2360.skip5	LDA &78
2370	PHA
2380	LDA &79
2390	PHA
2400	LDA &7A
2410	PHA
2420	LDA &7B
2430	PHA
2440	INC &76
2450	JSR down

2460	JMP loop4
2470	
2480.up	LDA &71
2490	CMP #&30
2500	BCC up2
2510	LDA &70
2520	AND #7
2530	BNE up1
2540	LDA &70
2550	SEC
2560	SBC #&79
2570	STA &70
2580	LDA &71
2590	SBC #2
2600	STA &71
2610	RTS
2620.up1	DEC &70
2630.up2	RTS
2640	
2650.down	LDA &71
2660	BMI down2
2670	LDA &70
2680	AND #7
2690	CMP #7
2700	BNE down1
2710	LDA &70
2720	CLC
2730	ADC #&79
2740	STA &70
2750	LDA &71
2760	ADC #2
2770	STA &71
2780	RTS
2790.down1	INC &70
2800.down2	RTS
2810	
2820.left	LDA &73
2830	BMI left1
2840	ASL &72
2850	BCC left1
2860	LDA #1
2870	STA &72
2880	DEC &73
2890	LDA &70

```
2900      SEC
2910      SBC #8
2920      STA &70
2930      LDA &71
2940      SBC #0
2950      STA &71
2960.left1 RTS
2970
2980.right LDA &73
2990      CMP #80
3000      BEQ right1
3010      LSR &72
3020      BCC right1
3030      LDA #128
3040      STA &72
3050      INC &73
3060      LDA &70
3070      CLC
3080      ADC #8
3090      STA &70
3100      LDA &71
3110      ADC #0
3120      STA &71
3130.right1 RTS
3140
3150.backgr LDA &73
3160      BMI off
3170      CMP #80
3180      BEQ off
3190      LDA &71
3200      BMI off
3210      CMP #&30
3220      BCC off
3230      LDY #0
3240      LDA (&70),Y
3250      AND &72
3260      BNE off
3270      LDA #0
3280      RTS
3290.off  LDA #1
3300      RTS
3310]
3320NEXT
3330ENDPROC
```

Notice that in this form this program takes up nearly 5K of memory. Disc users, in particular, may have trouble using this routine (though if it is typed in exactly as above it should just fit). If you need a shorter version you can just take out all the spaces and put more than one command per line and this should approximately halve the length of the assembly code.

Fill routines are not the most efficient way of filling large areas with colour. Where possible use the triangle plot command for large areas and keep the fill routine for filling small areas. If used to fill the whole screen, the routine above will take just over 30 seconds. However, for our example shape it only takes about four seconds.

SCREEN DUMPS

Modern dot-matrix printers can produce high-resolution graphics far in excess of the maximum resolution of the BBC Micro. It would seem sensible then to have a means of making a 'hard copy' of the entire contents of the screen. Such a copy of the screen is called a *screen dump*.

There are a number of screen dump programs around for various printers; however, they all tend to be slow. Many of them claim to be able to handle dumps of any size and of any mode with one program. This is fine, but to get a program to do this means sacrificing speed and performance. In this chapter we are going to look at a few examples of specialised screen dump programs. The advantage of these is that the top speed of the printer can be used.

The programs in this chapter are designed to be used with an Epson dot-matrix printer such as the FX80 or RX80. The MX range of printers will not handle the very-high-resolution dumps, but will handle, with only very slight modifications, the first few programs in this chapter. It should be possible to adapt some of the programs to run on other makes of printer. Before reading on, you might like to read the relevant section of your printer manual.

A simple BASIC dump

The first thing we should look at is a simple example—a Mode 4 screen dump about 4.5 in by 3.5 in.

We can consider the printer head as a vertical column of eight dots. The head is moved from left to right across the paper to produce eight rows of dots which make up a horizontal band. To print this band of dots, we need to send a series of codes to the

printer to tell it what vertical spacing between each band to use and how many dots make up a horizontal row. Then we send one byte for each vertical column of eight dots where each bit of that byte represents a dot (bit 7 is at the top, bit 0 is at the bottom). But this immediately presents us with a problem as the Mode 4 screen is stored with each byte representing a *horizontal* row of eight pixels (one bit represents one pixel in Mode 4).

To solve this, let's consider how we would dump one character (eight-by-eight pixels) to the printer. Let's assume that the character is placed at the top left-hand corner of the screen. We can use a BASIC program for this as it is only an experiment.

First we need to go into Mode 4 and turn off the cursor. Next we need to print the character (say, an A) at the top left-hand corner of the screen.

```
10 MODE4
20 VDU23,1,0;0;0;0;
30 P."A"
```

Now we must set-up the printer. The first thing is that we don't want the printer to automatically print a line-feed after every carriage return. This is because line-feed feeds the paper upwards by more than eight dots and we want each band (eight dots high) to butt up against the previous one. To do this we use *FX6,10 to disable the printer line-feed. Note that some printers are set-up so that they automatically print a line-feed whenever a carriage return is sent. If your printer is set-up to do this then the setting must be altered. This is done on Epson printers by changing the position of a small switch inside the printer. The printer manual gives details of how to do this. If you don't do this then the printer will split up your screen dumps into bands with gaps between them.

We then need to turn the printer on with VDU2. While we are doing this we can also tell the printer to go into a graphics mode where dots are printed horizontally $\frac{1}{80}$ of an inch apart. As the dots are automatically printed $\frac{1}{72}$ of an inch apart vertically in all graphics modes this will make our eight-by-eight

pixel character come out approximately square. We also need to tell the printer that we are going to send it eight bytes of graphics. The codes to do this are 27,42,4,8,0. Here, the first three numbers put the printer into the desired graphics mode and the last two are the number of bytes of graphics that we are sending to it. Remember that it is up to you to find out the correct control sequences for your make, model and mark of printer. To resume:

```
40 *FX6,10
50 VDU2,1,27,1,42,1,4,1,8,1,0
```

Notice that we use VDU1,X so that the data doesn't appear on the screen. Now we are ready to dump the character. Our letter A is stored on the screen as eight horizontal bytes (stored in addresses &5800 to &5807).

The pattern for A

Bit	7	6	5	4	3	2	1	0
&5800			●	●	●	●		
&5801		●	●			●	●	
&5802		●	●			●	●	
&5803		●	●	●	●	●	●	
&5804		●	●			●	●	
&5805		●	●			●	●	
&5806		●	●			●	●	
&5807								

To suit the printer, we need to code the A into eight vertical bytes. To do this we must set-up a bit mask which, using AND, will first mask out everything but bit 7, the leftmost bit, of each byte; then bit 6; and so on until all eight columns have been printed. We can store the current value of this mask in the variable A%—the initial value must be 128 to mask out everything but bit 7. We can use this mask on each of the eight bytes of the character in turn, to extract all the leftmost bits, making these into a vertical byte to send to the printer. Then we can set the mask to 64 to extract the next column, and so on until all eight columns have been printed.

For each column we first set a variable, say B%, to

zero. We then check the relevant bit of &5800 using the mask. If it is set then we add 1 to B%. We can then check the relevant bit of &5801, and so on. We want the top bit of the column (from &5800) to be stored in bit 7 of B%, and so on. If, before each byte is checked with the mask, we shift B% left one bit by multiplying it by two, and then we add 1 if the bit is set, then, after all the eight bytes have been checked the first bit we extracted (from &5800) will be in bit 7; the next bit (from &5801) will be in bit 6; and so on.

When we have done this to get the first column, we send B% to the printer and divide A% by two to move the mask right one pixel for the next column. We go on doing this until A% has reached zero and we have thus printed eight columns. Finally, we must send a carriage return to the printer and turn the printer off.

```
45 P%=&5800
60 A%=128
70 B%=0
80 FOR Y%=0 TO 7
90 B%=B%*2
100 IF P%?Y% AND A% THEN B%=B%+1
110 NEXT
120 VDU1,B%
130 A%=A%/2
140 IF A%>0 THEN 70
150 VDU1,13,3
```

This will dump one character from the screen. Notice that, because we have used a separate variable P% to store the address of the first byte of the character (line 45), we can print any character anywhere on the screen by simply changing P%. So, to print a whole band (which corresponds to a line of text on the screen) we need only alter the initial setting of the printer to say that we are going to send 40 characters of eight bytes each, or &140 bytes (line 50 below), add eight to P% after each character has been sent, and repeat this 40 times.

```
50 VDU2,1,27,1,42,1,4,1,&40,1,&1
```

```

55 FORX%=1T040
143 P%=P%+8
147 NEXT

```

To make a complete screen dump we now need to feed the paper upwards by eight dots (each dot takes up $\frac{1}{72}$ of an inch vertically) to print the next band. The Epson printers provide a command that will feed the paper upwards by $\frac{n}{216}$ of an inch. We want to feed by $\frac{8}{72}$ so n has to be 24. Because of the way the screen is laid out, P% already points to the start of the next band when we get to line 150; so, if P% is less than &8000 (the end of the screen), then we can go back to line 50 and print the next band down.

```

150 VDU1,13,1,27,1,74,1,24
160 IFP%<&8000THEN50
170 VDU3

```

This program can then be added to an existing program, or a screen can be *SAVED on disc and this routine can be used as a separate dump program by *LOADing the screen at the beginning.

A machine code equivalent

The next thing we need to do is to convert this BASIC into machine code. As in previous chapters, the above program has been purposely written to code into machine code easily.

The first thing we should do in the machine code version is disable all interrupts. This will help to speed up the program slightly. We can then disable the line-feeds.

```

10000 DEFPROCass
10010 DIMmc%150
10020 oswrch=&FFEE
10030 osbyte=&FFF4
10040 FORpass%=0T02STEP2
10050 P%=mc%
10060 [OPTpass%
10070 .dump SEI
10080 LDA #6

```

```
10090      LDX #10
10100      LDY #0
10110      JSR osbyte
```

Before we carry on any further, we should notice that almost every VDU command in the BASIC program was in the form VDU1,X. To save space it would be sensible to write a subroutine which outputs first a 1 then the byte we want to send to the printer.

```
20000 .out  PHA
20010      LDA #1
20020      JSR oswrch
20030      PLA
20040      JMP oswrch
```

This routine should be entered with the byte to be sent to the printer in the accumulator.

Going back to the main routine, we can set up P% in &70 and &71 and turn on the printer.

```
10120      LDA #0
10130      STA &70
10140      LDA #&58
10150      STA &71
10160      LDA #2
10170      JSR oswrch
```

Now we must initialise the printer for each band.

```
10180 .band LDA #27
10190      JSR out
10200      LDA #42
10210      JSR out
10220      LDA #4
10230      JSR out
10240      LDA #&40
10250      JSR out
10260      LDA #&1
10270      JSR out
```

We can use the X register to count the 40 characters that make up a horizontal band. We can use &72 to

store the bit mask, A%; &73 to store the byte sent to the printer, B%; and the Y register for the loop, Y%.

```
10280          LDX #40
10290 .char     LDA #128
10300          STA &72
10310 .column   LDA #0
10320          STA &73
10330          LDY #0
```

Next we need to shift &73 (B%) left. Then we must load a byte from the screen and mask it with &72. If there is a pixel in this position then we must set bit zero of &73. Then we can repeat the loop until Y is eight.

```
10340 .pixel    ASL &73
10350          LDA (&70),Y
10360          AND &72
10370          BEQ not
10380          LDA &73
10390          ORA #1
10400          STA &73
10410 .not      INY
10420          CPY #8
10430          BNE pixel
```

We can then send the column of eight pixels stored at &73 to the printer.

```
10440          LDA &73
10450          JSR out
```

Next we must shift the mask right one bit and repeat until we have dumped eight columns or a complete text character.

```
10460          LSR &72
10470          BNE column
```

Now we must add eight to &70 and &71 for the next character position and repeat back to the character routine 40 times to print one complete band.

```

10480      LDA &70
10490      CLC
10500      ADC #8
10510      STA &70
10520      BCC skip
10530      INC &71
10540      .skip DEX
10550      BNE char

```

Lastly we need to feed the paper upwards and repeat the band routine until the content of &71 is &80 or bigger. Then we can turn the printer off, enable interrupts and end the subroutine.

```

10560      LDA #13
10570      JSR out
10580      LDA #27
10590      JSR out
10600      LDA #74
10610      JSR out
10620      LDA #24
10630      JSR out
10640      LDA &71
10650      BPL band
10660      LDA #3
10670      JSR oswrch
10680      CLI
10690      RTS
20050 ]
20060 NEXT
20070 ENDPROC

```

And that's it.

As an example, try adding the following lines at the beginning of the assembly code.

```

10 MODE4
20 VDU23,1,0;0;0;0;29,640;512;
30 PROCass
40 S=1.1
50 P%=4
60 FORA=0TOPI*21STEPPI/20
70 PLOTP%,636*SIN(A),508*COS(A*S)

```

```
80 P%=5
90 NEXT
100 CALL dump
110 END
```

Mode 4 is very similar to Mode 0. To convert this program to run in Mode 0 only six lines need be changed.

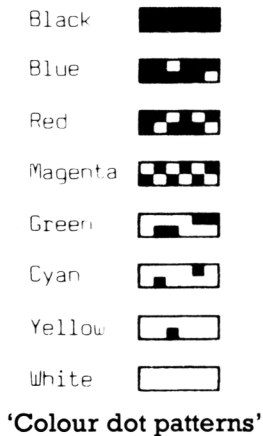
```
10 MODE0
10140 LDA #&30
10220 LDA #1
10240 LDA #&80
10260 LDA #2
10280 LDX #80
```

Notice that the Epson does not provide a graphics mode that has 160 pixels to the inch horizontally, so we have to use the closest mode; this gives only 120 pixels to the inch. This has the effect of stretching the screen dump slightly horizontally.

A colour-as-tone dump

So far we have only looked at two-colour dumps. What happens if we want to dump a 16-colour, Mode 2 screen? Most people don't own expensive colour printers, so let's find a way of representing colours on a black-and-white printer. It is easiest to do this with shades. Unfortunately, a dot-matrix printer won't automatically print greys for us. To get the illusion of shades we need to print a pattern of dots with more dots per square inch for the darker colours than for the lighter ones. For dumping in Mode 2 we will first ignore flashing colours and then represent each pixel by an imaginary box containing 12 dots arranged as two rows (height) of six columns (width). This arrangement is chosen because the pixels in Mode 2 are not square but rectangular—they are thin horizontal dashes rather than dots—so the printer must represent each pixel by a pattern of dots that is rectangular.

The pattern of dots for each colour (chosen to look as close to a solid block of colour as possible) is as follows.



This means that, as a Mode 2 screen is 160 pixels by 256 pixels, we will dump 960 (=6*160) by 512 (=2*256) dots on the printer.

Before we can do anything, we must know how a Mode 2 screen is laid out in the memory. As Mode 2 is a 16-colour mode, the computer must use four bits to store the colour of each pixel of the screen. Thus it can store two pixels ($2 \times 4 = 8$ bits) in one byte. If there are 40960 (160×256) pixels on the screen, this means that to store the complete screen we need 20480 bytes or 20K.

The two pixels stored in each byte are always adjacent to each other and appear in a horizontal line on the screen. We would expect to find that the first 80 consecutive bytes (=160 pixels) of the screen memory make up the top row of pixels. However, this unfortunately is not so. The screen memory is organised in terms of character positions. One character is made up of 64 (8×8) pixels. This means that 32 bytes of screen memory are needed to make up each character. Horizontally, these are split up into four columns, each two pixels (or one byte) wide. Each column is then made up vertically of eight consecutive bytes.

0	8	16	24
1	9	17	25
2	10	18	26
3	11	19	27
4	12	20	28
5	13	21	29
6	14	22	30
7	15	23	31

The order of the bytes that make up one text character.

The set of 32-byte characters is laid out, as would be expected, in 32 rows of 20 characters. So, for our purposes we can look at the screen as divided into 32 rows. Each row is stored as 640 bytes and consists of 80 columns, each column being eight bytes tall. The first byte of the screen is stored at &3000 and the last at &7FFF.

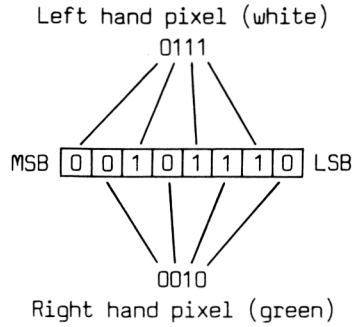
Of course, all this changes for different modes.

**Mode 2 screen
organisation**

&3000	&3008					&3270	&3278
&3001	&3009					&3271	&3279
&3002	&300A					&3272	&327A
&3003	&300B					&3273	&327B
&3004	&300C					&3274	&327C
&3005	&300D					&3275	&327D
&3006	&300E					&3276	&327E
&3007	&300F					&3277	&327F
&3280	&3288					&34F0	&34F8
&3281	&3289					&34F1	&34F9
&3282	&328A					&34F2	&34FA
&3283	&328B					&34F3	&34FB
&3284	&328C					&34F4	&34FC
&3285	&328D					&34F5	&34FD
&3286	&328E					&34F6	&34FE
&3287	&328F					&34F7	&34FF

Finally, we must examine how the two Mode 2 pixels are stored in each screen memory byte. Again, we would expect the most significant four bits of the byte to hold the colour of the first pixel and the least significant four bits to hold the colour of the second pixel. Yet again, things are slightly more complex. The two four-bit numbers which represent the colours of the two pixels are 'interleaved'—first a bit from pixel one (the left-hand pixel) then a bit from pixel two (the right-hand pixel) then a bit from pixel one again, and so on.

**Mode 2 pixels:
layout**



It is obvious from this diagram that ours is not an easy task. Remember that we need to send graphics to the printer in bands of eight vertical dots (corresponding to the eight vertical dots on the printer head). This means that we need to group four screen pixels (each two dots high) above each other. Let's first of all assume that we have found the colours of each of these pixels (ignoring the most significant bit to exclude flashing colours) and we have stored the four of them, from top to bottom, in addresses &74 to &77. Now let's try and write a routine to dump these four pixels to the printer.

As often with machine code, the easiest way to do this is with a look-up table. This will contain data on the pattern of dots for each colour. However, we need to think carefully about how to arrange this information in a table. As the printer deals in vertical columns, the sensible way to split up the 12 dots per pixel is into six columns, devoting one byte of table for each column. Although we are wasting memory by only using two bits in each byte of the table, it will speed up the program if we use one byte of table for each column.

We now have to decide how to order these columns. The obvious way is to store them as six bytes for black followed by six bytes for red, et-cetera. However, we must consider how we are going to address this table. To print the four vertical pixels we are going to send six bytes to the printer. The first byte will be made up of the first columns of each of the relevant colour patterns, the second byte will be made up of the second columns of the patterns, and so on. So, to send the first byte to the

printer we are going to need access to the eight bytes that contain the first columns of each of the eight colour patterns. Then for the second byte we will need the eight bytes of the second columns and so on. Thus it makes sense to store the eight bytes that contain the first columns of each of the colour patterns, grouped together in the table.

To make the program easier we can store the data in groups according to the colours but just read them into the table in our preferred order.

```
10000 DEFPROCass
10010 DIMmc%300,D%47
10020 RESTORE20000
10030 FORA%=0TO7
10040 FORB%=0TO5
10050 READD%?(A%+B%*8)
10060 NEXT,

20000 DATA3,3,3,3,3,3:REM Black
20010 DATA3,2,1,3,1,2:REM Red
20020 DATA0,1,1,0,2,2:REM Green
20030 DATA0,0,1,0,0,0:REM Yellow
20040 DATA3,3,1,3,3,2:REM Blue
20050 DATA1,2,1,2,1,2:REM Magenta
20060 DATA0,1,0,0,2,0:REM Cyan
20070 DATA0,0,0,0,0,0:REM White
```

Now the routine to print four vertical pixels should be relatively easy. First we are going to need, from the previous program, the routine for sending a byte to the printer.

```
10070 oswrch=&FFEE

14000 .out      PHA
14010          LDA #1
14020          JSR oswrch
14030          PLA
14040          JMP oswrch
```

Now we can start. The first job is to set up the address of the start of the table in &7A and &7B. Then we can use post-indexed indirect addressing

to look up the bytes for the first column of each pixel by setting the Y register to the colour number. Then when we have sent this byte to the printer we can add eight to the contents of &7A and &7B so that, taken together, they point to the address of the eight second-column bytes. We can use the X register to count the number of bytes we have sent to the printer.

```
13000 .print   LDA #0%MOD256
13010         STA &7A
13020         LDA #0%DIV256
13030         STA &7B
13040         LDX #6
```

Next we can set Y to the contents of &74 (the top pixel colour) and load in the two bits for the top two dots of the first column. These must eventually be the most significant two bits of the byte so we must shift them left. We shift left the bytes we are calculating, by two bits each time a pixel has been calculated, and so save a lot of effort.

```
13050 .prloop  LDY &74
13060         LDA (&7A),Y
13070         ASL A
13080         ASL A
13090         STA &79
13100         LDY &75
13110         LDA (&7A),Y
13120         ORA &79
13130         ASL A
13140         ASL A
13150         STA &79
13160         LDY &76
13170         LDA (&7A),Y
13180         ORA &79
13190         ASL A
13200         ASL A
13210         STA &79
13220         LDY &77
13230         LDA (&7A),Y
13240         ORA &79
13250         JSR out
```

We have now sent the first of our six bytes to the printer and all that remains is to add eight to &7A and &7B and go back for the next byte.

```
13260          LDA &7A
13270          CLC
13280          ADC #8
13290          STA &7A
13300          BCC skip2
13310          INC &7B
13320 .skip2   DEX
13330          BNE prloop
13340          RTS
```

This is all very well, but we still need to find out the four colours from the screen. We could use the operating system command that is equivalent to the BASIC function POINT, but this is slow and it is faster to work it out ourselves.

Let's now look at the main program and let's assume that we have already written a routine that will dump a whole band of Mode 2 graphics, four pixels high, given the address of the top left-hand corner of the line on the screen stored in &70 and &71.

Firstly, we need to disable interrupts and the printer line-feed, and turn the printer on.

```
10080 FORpass%=0T02STEP2
10090 P%=mc%
10100 [OPTpass%
10110 .dump   SEI
10120          LDA #6
10130          LDX #10
10140          LDY #0
10150          JSR &FFF4
10160          LDA #2
10170          JSR oswrch
```

Next, we need to store the address of the top left-hand corner of the screen in &70 and &71. Then to print the top half of a text line we need only call our *band* routine.

```
10180      LDA #0
10190      STA &70
10200      LDA #&30
10210      STA &71
10220      .textlin JSR band
```

To print the bottom half we need to add four to the start address and call *line* again.

```
10230      LDA &70
10240      CLC
10250      ADC #4
10260      STA &70
10270      BCC skip
10280      INC &71
10290      .skip JSR band
```

To move on to the next line we need to add &280 to the address of the previous line, but we have already added four so we only need add &27C. If the answer is less than &8000 then we can go back and dump the next text line. Otherwise we must turn off the printer, re-enable the interrupts and end the routine.

```
10300      LDA &70
10310      CLC
10320      ADC #&7C
10330      STA &70
10340      LDA &71
10350      ADC #2
10360      STA &71
10370      BPL textlin
10380      LDA #3
10390      JSR oswrch
10400      CLI
10410      RTS
```

Now all we have to do is make our assumptions concrete and actually write *band*.

Firstly, we need to make a copy of &70 and &71 in &72 and &73. Then we need to initialise the printer to accept a 960-column band of graphics.

```

11000 .band    LDA &70
11010          STA &72
11020          LDA &71
11030          STA &73
11040          LDA #27
11050          JSR out
11060          LDA #42
11070          JSR out
11080          LDA #1
11090          JSR out
11100          LDA #&C0
11110          JSR out
11120          LDA #3
11130          JSR out

```

Now we need to count the 80 groups of pixels that we must send to the printer (160 pixels per line and two pixels per byte). As we need both the X and Y registers we use location &78 to hold a count.

```

11140          LDA #80
11150          STA &78

```

Before we carry on, we need two routines that will extract the colours of the two pixels from a byte. Let us deal first with the left-hand pixel stored in the odd-numbered bits. We can arrange for the three bits of interest to us to be in bits 2, 4 and 6 (least significant to most significant) of the accumulator, using ASL once; then we can do two left-shifts to transfer the most significant bit in bit 6 into the carry flag. Then we can ROL the carry flag into a memory byte previously set to zero (&79). This leaves the next bit in bit 6 of the accumulator so we can repeat this a further two times to leave our three bits in bits 0-2 of location &79. If this sounds complicated, try following the code through.

```

12000 .colour1 LDA (&72),Y
12010 .colour  ASL A
12020          INY
12030          LDX #0
12040          STX &79
12050          LDX #3
12060 .coloop  ASL A

```

12070	ASL A
12080	ROL &79
12090	DEX
12100	BNE coloop
12110	LDA &79
12120	RTS

The INY command loads the next byte down on the next call of this routine. If we want to extract the colour of the other pixel we need only load the byte into the accumulator and shift it left one bit so that the bits we want occupy the same positions as for the other pixel. Then we can jump to the label *colour*, conveniently placed in the previous routine, to finish the job.

12130	.colour2 LDA (&72),Y
12140	ASL A
12150	JMP colour

Now we can go back to *band* where we were just about to dump a line of graphics.

Firstly, we must set Y to zero so that we are ready to load the top byte. We can then call *colour1* to find the colour of the left-hand pixel and store this information in &74 ready for *print*. We can then do the same for the other four pixels and call *print*.

11160	.column LDY #0
11170	JSR colour1
11180	STA &74
11190	JSR colour1
11200	STA &75
11210	JSR colour1
11220	STA &76
11230	JSR colour1
11240	STA &77
11250	JSR print

We do the same for the four right-hand pixels.

11260	LDY #0
11270	JSR colour2
11280	STA &74

```
11290      JSR colour2
11300      STA &75
11310      JSR colour2
11320      STA &76
11330      JSR colour2
11340      STA &77
11350      JSR print
```

Now, to print the next set of four bytes, we need to add eight to &72 and &73 and repeat until we have printed all 80 groups of pixels.

```
11360      LDA &72
11370      CLC
11380      ADC #8
11390      STA &72
11400      BCC skip1
11410      INC &73
11420      .skip1 DEC &78
11430      BNE column
```

Lastly we need to feed the paper and return.

```
11440      LDA #13
11450      JSR out
11460      LDA #27
11470      JSR out
11480      LDA #74
11490      JSR out
11500      LDA #24
11510      JMP out
```

```
15000 ]
15010 NEXT
15020 ENDPROC
```

At last we have finished. Try the following example to check that the routine works.

```
10 MODE2
20 VDU23,1,0;0;0;0;
30 PROCass
40 COLOUR135
50 CLS
```

```
60 COLOUR128
70 RESTORE180
80 P%=4
90 FORA%=0TO8
100 READB%
110 GCOLO,B%
120 MOVEO,0
130 PLOTP%,1000*SIN(A%*PI/16),1000*COS
    (A%*PI/16)
140 P%=85
150 NEXT
160 CALLdump
170 END
180 DATA0,0,4,1,5,2,6,3,7
```

A miniature dump

So far, we have not printed anything particularly revolutionary. However, with many modern printers it is possible to produce screen dumps that put to shame what we have looked at so far.

Some Epson and compatible printers have a graphics mode in which 240 dots to the inch can be printed horizontally. Unfortunately, adjacent dots cannot be printed as the print head needs time to recover after each dot has been printed. To overcome this we can print each line *twice*, first printing all the even-numbered dots then all the odd-numbered dots. This still leaves us with the vertical resolution—fixed by the spacing of the print wires in the print head—of 72 dots to the inch. This is more difficult to overcome.

To solve this problem we use another special feature of Epson and compatible printers—they can feed the paper relatively accurately by down to $\frac{1}{216}$ of an inch! This means that we can inter-leave three columns of dots at a spacing of $\frac{1}{72}$ of an inch. The diagram on page 200 shows this. The pixels printed on the first pass are numbered 1, etcetera. This means that the maximum resolution of the printer is 240 dots per inch horizontally and 216 dots per inch vertically. This makes a staggering 51840 dots per square inch! At this resolution we should be able to dump a full Mode 4 screen in about 1.6 square inches—so that's what we are going to do!

Dot interleaving

```
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
1 2 1 2 1 2 1 2 1 2
3 4 3 4 3 4 3 4 3 4
5 6 5 6 5 6 5 6 5 6
```

It is very difficult to try and use direct screen access to dump the screen in this example (though you are welcome to try if you are feeling masochistic) so we will use OSWORD call 9 (the operating system equivalent of POINT).

The first thing we need, then, is a routine to set-up a parameter block and call OSWORD 9. Let us state that the X coordinate should be stored in &70 and &71 and the Y coordinate in &72 and &73. This means that our OSWORD parameter block can be taken as starting at &70 and the colour will be returned at &74. The first job is to save the X and Y registers and set-up the three registers for the call.

```
20000 .point STX &76
20010 STY &77
20020 LDA #9
```

```
20030      LDY #0
20040      LDX #&70
20050      JSR &FFF1
```

Now we can reload the X and Y registers with their original values (saved at &76 and &77). We also need to check if the point was on the screen. This is because there are 256 pixels vertically on the screen and we are printing in lots of 24. Now, 24 into 256 doesn't go, so the last line is going to drop off the bottom and when this happens *point* must return black. The OSWORD call we're using will return 255 so we need only check that the colour is positive; and, if not, return zero.

```
20060      LDX &76
20070      LDY &77
20080      LDA &74
20090      BPL point1
20100      LDA #0
20110      .point1 RTS
```

We also need our old faithful routine, *out*.

```
20120      .out    PHA
20130      LDA #1
20140      JSR oswrch
20150      PLA
20160      JMP oswrch
```

Now, the main routine. We reset the graphics windows and origin, as we are using POINT; otherwise, we might end up dumping the wrong part of the screen. Also, we can disable interrupts.

```
10000 DEFPROCass
10010 DIMmc%300
10020 oswrch=&FFEE
10030 FORpass%=0TO2STEP2
10040 P%=mc%
10050 [OPTpass%
10060      .dump    SEI
10070      LDA #26
10080      JSR oswrch
```

Next we must disable printer line-feeds and turn on the printer.

```
10090      LDA #6
10100      LDX #10
10110      LDY #0
10120      JSR &FFF4
10130      LDA #2
10140      JSR oswrch
```

Now we must set the Y coordinate of the top pixel of the screen in &78 and &79 ready to work down the screen.

```
10150      LDA #&FC
10160      STA &78
10170      LDA #3
10180      STA &79
```

For the moment let's assume we already have a routine *band* that prints one pass of the print head, i.e. given the coordinates of the top left-hand pixel of a band (320 pixels by 24 pixels) it prints one sixth of the dots on that band. The X coordinate needs to be in &70 and &71 and the Y coordinate in &78 and &79. Also, we need to tell the routine whether it is printing odd-numbered or even-numbered dots, horizontally, on that pass. We can do that by supplying a mask in &7A—either 255 for the even dots or zero for the odd ones. The reason for this will become obvious in a moment. Armed with this routine, we can finish the main routine.

The first job, as we are going to print three interlaced passes vertically, is to set the X register to count these.

```
10190 .loop1  LDX #3
```

Next, we need to set up the X coordinate as zero and the mask as 255 for the even pixels.

```
10200 .loop2  LDA #0
10210      STA &70
10220      STA &71
```

```
10230      LDA #255
10240      STA &7A
10250      JSR band
```

Next, for the odd pixels we need to set the X coordinate to four and the mask to zero.

```
10260      LDA #4
10270      STA &70
10280      LDA #0
10290      STA &71
10300      STA &7A
10310      JSR band
```

Now we must feed the paper $\frac{1}{8}$ of an inch ready for the next interleaved set of pixels. We also need to move the Y coordinate down one pixel and then repeat the whole process.

```
10320      LDA #27
10330      JSR out
10340      LDA #74
10350      JSR out
10360      LDA #1
10370      JSR out
10380      LDA &78
10390      SEC
10400      SBC #4
10410      STA &78
10420      BCS skip3
10440      DEC &79
10450      .skip3 DEX
10460      BNE loop2
```

We have now printed a whole band 24 dots high and need to feed the paper onward 24 dots (less the three we have already fed it).

```
10470      LDA #27
10480      JSR out
10490      LDA #74
10500      JSR out
10510      LDA #21
10520      JSR out
```

We then need to move the Y coordinate down 24 pixels (less the three we have already moved it); and, if the Y coordinate is still on the screen, we must go back to dump the next band.

```
10530      LDA &78
10540      SEC
10550      SBC #84
10560      STA &78
10570      LDA &79
10580      SBC #0
10590      STA &79
10600      BPL loop1
```

All that remains is to disable the printer, re-enable the interrupts again and exit.

```
10610      LDA #3
10620      JSR oswrch
10630      CLI
10640      RTS
```

Now we need to write *band*. The first thing this routine needs to do is to set-up the printer to receive 320 bytes of graphics.

```
11000 .band  LDA #27
11010      JSR out
11020      LDA #42
11030      JSR out
11040      LDA #3
11050      JSR out
11060      LDA #&40
11070      JSR out
11080      LDA #1
11090      JSR out
```

Next we need to make a temporary working copy of the Y coordinate in &72 and &73 for *point* to use.

```
11100 .line1 LDA &78
11110      STA &72
11120      LDA &79
11130      STA &73
```

We now need to work out a byte. Again, we are going to use the technique of shifting a byte of memory left while setting bit one to the colour of a pixel and repeating this eight times. So first we need to set the byte (&75) to zero. Then we need the Y register to count the eight times. The first job inside the loop is to shift &75 left a bit. Then we can use *point* to return the colour of the pixel. If it is zero then we can leave &75 alone as we have already set all the bits to zero. Otherwise, we need to set bit zero to one. As this will always be zero to start with we can simply use the command INC to set it to one.

```
11140          LDA #0
11150          STA &75
11160          LDY #8
11170 .line2   ASL &75
11180          JSR point
11190          BEQ skip1
11200          INC &75
11210 .skip1
```

Next we need to move down three pixels to allow for the interleave, and repeat the process.

```
11210 .skip1   LDA &72
11220          SEC
11230          SBC #12
11240          STA &72
11250          BCS skip2
11260          DEC &73
11270 .skip2   DEY
11280          BNE line2
```

We are now ready to send a byte to the printer. However, if &7A is set to 255, then we want to send the byte followed by a zero; if it is set to zero, then we want to send a zero followed by the byte. So, as the first byte to send to the printer, we can use (&75 AND &7A). For the second byte we can use (&75 AND (&7A EOR 255)).

```
11290          LDA &7A
11300          AND &75
```

```
11310      JSR out
11320      LDA &7A
11330      EOR #255
11340      AND &75
11350      JSR out
```

Now all that remains to do is move on to the next-pixel-but-one horizontally by adding eight to the X coordinate; and then, if the answer is less than 1280 or &500, go back and send the next pair of bytes. If the end of the line has been reached then we must send a carriage return and exit from the routine.

```
11360      LDA &70
11370      CLC
11380      ADC #8
11390      STA &70
11400      LDA &71
11410      ADC #0
11420      STA &71
11430      CMP #5
11440      BNE line1
11450      LDA #13
11460      JMP out

































25000 ]
25010 NEXT
25020 ENDPROC
```

We now have a finished program. To try it out you can use the example program from the large Mode 4 dump routine.

Now that we have discovered the maximum resolution of the printer it is well worth going back to the Mode 2 dump. Let us consider using pixels represented as six dots by three dots. If we dump a Mode 2 screen at the highest resolution using this system then it will be about 4 inches by 3.5 inches. Also, we should be able to make a grey scale that will handle 16 levels of brightness. This means that we can use each of the 16 colours that the BBC Micro will handle in Mode 2. Obviously we can't represent flashing colours so the best way to make use of this is not to

try and print exactly what is on the screen but to represent each colour from 0 to 15 as a shade with 0 darkest and 15 brightest.

It turns out that with six-by-three dots at the highest resolution the third row can always be left blank and we still get a good range of colours. This speeds printing up as we will only have to print four interleaved lines. But if we try to print the same patterns for each pixel, one above the other, they will tend to produce visible vertical lines. So we will use *two* sets of patterns and use alternate sets for alternate rows of pixels. The patterns we will use are as below:

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

As we are going to dump all the top rows then all the bottom rows, it would seem sensible to block all the top rows in one group and all the bottom rows in one group. Within each group we can group the entries into all the column ones then all the column twos, etcetera. We can store the two alternate sets of patterns, in this format, one after the other. We are going to build up the bytes to send to the printer by shifting them left and ORing them with 1 or 0. For this reason it will be easiest if we use one whole byte of a table to store each dot of the patterns. These bytes will either be 1 for a dot or 0 for no dot. The data looks as set out below.

20000 DATA1,1,1,0,0,0,1,1
20010 DATA1,1,0,0,0,0,0,0
20020 DATA1,1,1,1,1,1,1,0
20030 DATA0,0,0,0,0,0,0,0
20040 DATA1,0,0,0,0,0,0,0
20050 DATA0,0,1,0,0,0,0,0
20060 DATA1,1,1,1,1,1,0,0
20070 DATA0,0,0,1,0,0,0,0
20080 DATA1,1,0,0,0,0,0,1
20090 DATA1,0,0,1,1,0,0,0
20100 DATA1,1,1,1,1,0,0,0
20110 DATA0,0,0,0,0,0,0,0
20120 DATA1,1,1,1,1,1,0,0
20130 DATA0,0,0,0,0,0,0,0
20140 DATA1,1,0,0,0,0,0,0
20150 DATA0,0,0,0,0,0,0,0
20160 DATA1,1,1,1,0,0,0,1
20170 DATA1,0,0,0,0,0,0,0
20180 DATA1,1,1,0,0,0,1,0
20190 DATA0,1,0,0,0,0,0,0
20200 DATA1,1,1,1,1,1,1,0
20210 DATA0,0,0,0,0,0,0,0
20220 DATA1,0,0,0,0,0,0,0
20230 DATA0,0,0,0,0,0,0,0
20240 DATA1,1,1,0,0,0,1,1
20250 DATA1,1,0,0,0,0,0,0
20260 DATA1,1,1,1,1,1,1,0
20270 DATA0,0,0,1,1,0,0,0
20280 DATA1,0,0,0,0,0,0,0
20290 DATA0,0,0,0,0,1,1,0
20300 DATA1,1,1,1,1,1,0,0
20310 DATA0,0,0,0,0,1,0,0
20320 DATA1,1,0,0,0,0,0,1
20330 DATA0,0,1,0,0,0,0,0
20340 DATA1,1,1,1,0,0,0,0
20350 DATA0,0,0,0,0,0,0,0
20360 DATA1,1,1,1,1,1,0,0
20370 DATA0,0,1,0,0,0,0,0
20380 DATA1,1,0,0,0,0,0,0
20390 DATA0,0,0,0,0,0,0,0
20400 DATA1,1,1,1,1,0,0,1
20410 DATA0,0,0,0,0,0,0,0
20420 DATA1,1,1,0,0,0,1,0
20430 DATA1,1,0,0,0,0,0,0

```
20440 DATA1,1,1,1,1,1,1,0
20450 DATA0,0,0,0,0,0,0,0
20460 DATA1,0,0,0,0,0,0,0
20470 DATA0,0,0,0,0,0,0,0
```

So our first job is to read this into a reserved area of memory.

```
10000 DEFPROCass
10010 DIMmc%1000,D%383
10020 oswrch=&FFEE
10030 FORA%=0T0383
10040 READD%?A%
10050 NEXT
10060 FORpass%=0T02STEP2
10070 P%=mc%
10080 [OPTpass%
```

We will again need our trusty routines, *point* and *out*. This time we don't need to worry about the point being off the screen as we are printing at eight pixels per band and eight goes into 256 exactly.

```
18000 .point STX &76
18010 STY &77
18020 LDA #9
18030 LDY #0
18040 LDX #&70
18050 JSR &FFF1
18060 LDX &76
18070 LDY &77
18080 LDA &74
18090 RTS
19000 .out PHA
19010 LDA #1
19020 JSR oswrch
19030 PLA
19040 JMP oswrch
19050 ]
19060 NEXT
19070 ENDPROC
```

Now for the main routine. As with the previous routine we need first of all to disable interrupts,

reset windows, disable printer line-feeds and turn the printer on.

```
10090 .dump    SEI
10100          LDA #26
10110          JSR oswrch
10120          LDA #6
10130          LDX #10
10140          LDY #0
10150          JSR &FFF4
10160          LDA #2
10170          JSR oswrch
```

Next we need to set up the Y coordinate of the top of the screen in &78 and &79.

```
10180          LDA #&FC
10190          STA &78
10200          LDA #3
10210          STA &79
```

Before we carry on, we need a routine to print one pass of the printer head. We can specify that on entry the Y coordinate of the top left-hand corner of the band is in &78 and &79; that the contents of &7B are zero for the top row and 96 for the bottom row (this allows us to add this to the table address to take care of which row we are printing); and that &7A contains a mask which is 255 for the even-numbered dots and zero for the odd-numbered ones.

The first job, as always, is to set the printer to the right graphics mode—here, quadruple-density with 960 (&3C0) dots across.

```
15000 .band    LDA #27
15010          JSR out
15020          LDA #42
15030          JSR out
15040          LDA #3
15050          JSR out
15060          LDA #&C0
15070          JSR out
15080          LDA #3
15090          JSR out
```

Next we must set the X coordinate to zero.

```
15100      LDA #0
15110      STA &70
15120      STA &71
```

Next, for each column of pixels we send to the printer, we need to set-up the address of the table in &7C and &7D. This will be D% plus the contents of &7B.

```
15130 .column LDA #D%MOD256
15140          CLC
15150          ADC &7B
15160          STA &7C
15170          LDA #D%DIV256
15180          ADC #0
15190          STA &7D
```

Now we are ready to send six bytes to the printer. We can use the X register to count the bytes. For each byte we need to make a temporary working copy of the contents of &78 and &79 in &72 and &73.

```
15200      LDX #6
15210 .byte  LDA &78
15220      STA &72
15230      LDA &79
15240      STA &73
```

We are going to work on the byte in &75 so we need to set it to zero for starters. Then we can count the bits we have worked on, with the Y register.

```
15250      LDA #0
15260      STA &75
15270      LDY #8
```

Now for every bit we calculate we first need to shift &75 left a bit. Then we must find out the colour of the pixel.

```
15280 .bit  ASL &75
15290      JSR point
```

We are going to use the Y register to point into the table so we need save the Y register at &76 until we have finished with the table. We have the colour of the pixel in the accumulator, so, by transferring it to the Y register we can use it directly to point into the table. However, if we are calculating an odd-numbered bit then we need to use the second set of shade patterns; these are 192 bytes further on in the table. We need to check bit 0 of the bit count which we have temporarily stored at &76; if it is one, we must add 192 to the Y register. Then we can load a byte from the table. At this point we have finished with the Y register and can reload its original bit count value.

```
15300          STY &76
15310          TAY
15320          LDA #1
15330          BIT &76
15340          BEQ skip1
15350          TYA
15360          CLC
15370          ADC #192
15380          TAY
15390 .skip1    LDA (&7C),Y
15400          LDY &76
```

This byte we need to OR with the byte we are calculating. We then need to move down a pixel and, if we haven't already finished the byte, go back and calculate the next bit.

```
15410          ORA &75
15420          STA &75
15430          LDA &72
15440          SEC
15450          SBC #4
15460          STA &72
15470          BCS skip2
15480          DEC &73
15490 .skip2    DEY
15500          BNE bit
```

Now we can send the byte to the printer. However,

we must only send alternate bytes so we must mask it with &7A and reverse the mask in &7A ready for the next byte.

```
15510      LDA &75
15520      AND &7A
15530      JSR out
15540      LDA &7A
15550      EOR #255
15560      STA &7A
```

Next we must move the start of the table to the next column. Then, unless we have printed all six, we must go back and print the next column.

```
15570      LDA &7C
15580      CLC
15590      ADC #16
15600      STA &7C
15610      BCC skip3
15620      INC &7D
15630      .skip3 DEX
15640      BNE byte
```

We have now dumped a column of eight pixels and can move on to the next column. If we have printed a whole line then we can send a carriage return and exit the routine.

```
15650      LDA &70
15660      CLC
15670      ADC #8
15680      STA &70
15690      LDA &71
15700      ADC #0
15710      STA &71
15720      CMP #5
15730      BNE column
15740      LDA #13
15750      JMP out
```

We can now finish off the main routine. For every text line, we must first set &7B to zero for the top row. Then, for each row, we must make two passes:

first with the mask set to 255, then with the mask set to zero.

```
10220 .textlin LDA #0
10230          STA &7B
10240 .row    LDA #255
10250          STA &7A
10260          JSR band
10270          LDA #0
10280          STA &7A
10290          JSR band
```

Next we must feed the paper a fraction and set &7B to 96 for the second row. By EXCLUSIVE ORing &7B with 96 we can use this to count the two rows.

```
10300          LDA #27
10310          JSR out
10320          LDA #74
10330          JSR out
10340          LDA #1
10350          JSR out
10360          LDA &7B
10370          EOR #96
10380          STA &7B
10390          BNE row
```

Now we can feed the paper up the rest of the line. Because the printer will not feed accurately at $\frac{1}{216}$ of an inch it tends to feed more than this, so the two feeds we have performed will have fed closer to $\frac{3}{216}$ of an inch. To correct for this we need to feed only a further $\frac{21}{216}$ of an inch. This may not be necessary on some printers, so you should experiment.

```
10400          LDA #27
10410          JSR out
10420          LDA #74
10430          JSR out
10440          LDA #21
10450          JSR out
```

All that remains is to move down to the next text line and repeat until we have dumped the whole screen;

then we turn off the printer, re-enable the interrupts, and exit in the usual fashion.

```
10460      LDA &78
10470      SEC
10480      SBC #32
10490      STA &78
10500      LDA &79
10510      SBC #0
10520      STA &79
10530      BPL textlin
10540      LDA #3
10550      JSR oswrch
10560      CLI
10570      RTS
```

To try out the dump routine, add the following lines to the assembly code:

```
10 MODE2
20 VDU23,1,0;0;0;0;
30 PROCass
40 COLOUR143
50 CLS
60 COLOUR128
70 P%=4
80 FORA%=0T016
90 GCOLO,A%-1
100 MOVE0,0
110 PLOTP%,1000*SIN(A%*PI/32),1000*COS
    (A%*PI/32)
120 P%=85
130 NEXT
140 CALLdump
150 END
```

If you want to produce larger pictures it is relatively easy to join two or more dumps together. We have only looked at a selection of the screen dumps that can be written but these should give you an idea of how to go about writing any others you may need.

SPRITE GRAPHICS

Most arcade games feature a series of animated characters which move around the screen. One or more of the characters are controlled by the player. On the BBC Micro, the only help the operating system gives to anyone trying to produce such graphics is in the provision of the user-definable character set. Using VDU23 it is easy to produce eight-by-eight pixel shapes which can be moved around the screen.

However, this system has its limitations. Firstly, the shape produced can only be in two colours, background and foreground; secondly, eight-by-eight is too small for most purposes; and thirdly, this method is far too slow for a fast action-packed arcade game.

The first and second problems can be solved by combining more than one character to make up an object, but this makes the animation even slower. It is slow because time is taken up by the characters having to be converted from the eight-byte format of the user-defined character to the form in which they are actually stored in the screen memory. Worse still, the way a character is stored varies between the different screen modes.

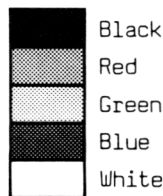
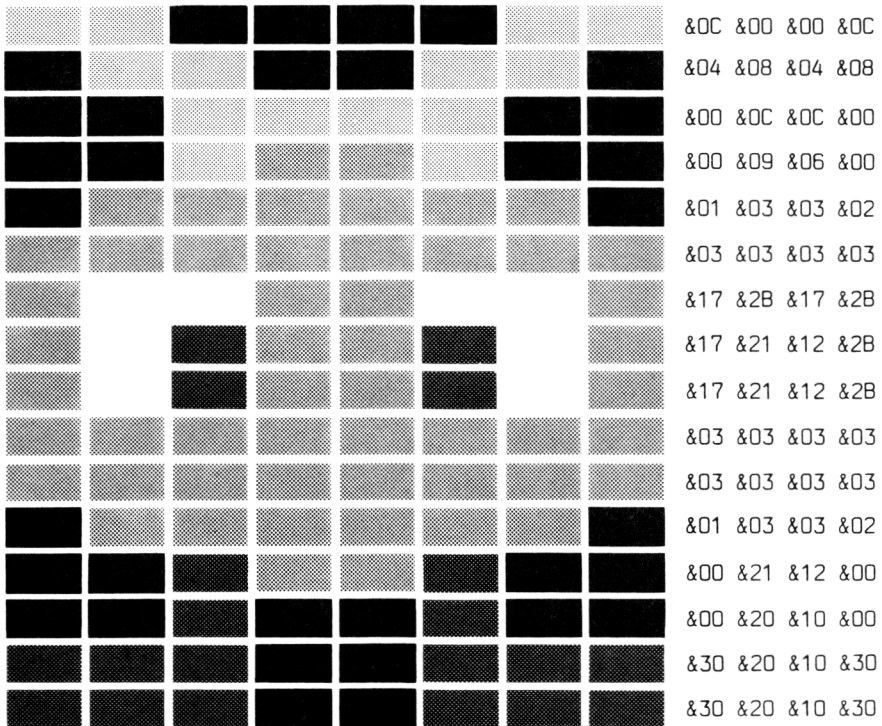
However, as most games will only use one graphics mode, it should be possible to code the character into the relevant format for that particular mode when writing the program. This ready-defined character would be able to contain all the colours available in the mode and could be any size. This shape could then be stored directly on the screen in a fraction of the time taken by the operating system to do the same job. These pre-defined characters are called SPRITES. As most arcade games work in Mode 2, I am going to show how to

use a complete sprite system from machine code in this mode.

Remember that the methods about to be detailed will not work across the Tube.

Before we embark on a complex machine code routine, we should try an experiment in BASIC—this, as we have seen, is always a good idea when writing machine code routines.

A sample sprite



A BASIC sprite routine

We are going to use the sprite shown above as an example. The coding for its storage as a Mode 2 sprite is shown. Because of the way in which the

screen is laid out this coding will only work if the sprite starts on the first pixel of a screen memory byte. That is, the furthest-left pixel of the sprite must be on an even-numbered pixel horizontally. If we wanted to place it a single pixel to the right or left, we would have to totally re-code it.

However, we can easily move the sprite left and right two pixels at a time; that way, we are moving it one byte at a time. If we made the sprite move this distance every fiftieth of a second (which is the rate at which the image on a TV or monitor is updated), the image would appear to be moving smoothly.

However, if we wanted the sprite to move slower than that, we would either have to put up with noticing that the sprite jumps two pixels at a time, or we would have to define two sprites, one in each position, and alternate between them. This is common practice in arcade games and very often the two sprites are slightly different. For example, it is quite effective to use two sprites of a man with his legs in different positions. This will make him appear to walk when the sprites are placed alternately on the screen.

For movement up and down, we need to place the bytes from the shape table into the screen memory in different positions. As we shall see, this is not too difficult. We can move the sprite up and down a pixel at a time without having to recode the sprite shape table.

The BASIC program below will place our example sprite in the top left-hand corner of the screen. The data statements at the end contain the coded data for the sprite laid out as above. Remember, this is an 8-by-16 sprite, so it is stored as 4 bytes (= 8 pixels) wide and 16 bytes high.

```
10 MODE2
20 VDU23,1,0;0;0;0;
30 FOR A% = 0 TO 1
40 FOR B% = 0 TO 7
50 FOR C% = 0 TO 3
60 READ D%
70 ?(&3000+A%*640+B%+C%*8)=D%
```

```

80 NEXT,,
90 GOTO90
20000 DATA&0C,&00,&00,&0C
20010 DATA&04,&08,&04,&08
20020 DATA&00,&0C,&0C,&00
20030 DATA&00,&09,&06,&00
20040 DATA&01,&03,&03,&02
20050 DATA&03,&03,&03,&03
20060 DATA&17,&2B,&17,&2B
20070 DATA&17,&21,&12,&2B
20080 DATA&17,&21,&12,&2B
20090 DATA&03,&03,&03,&03
20100 DATA&03,&03,&03,&03
20110 DATA&01,&03,&03,&02
20120 DATA&00,&21,&12,&00
20130 DATA&00,&20,&10,&00
20140 DATA&30,&20,&10,&30
20150 DATA&30,&20,&10,&30

```

Notice in line 70 that the first eight rows of the sprite are placed from address &3000 onwards and the second eight rows are placed 640 bytes further on because they are on the next text line. Notice also that the position *along* a row, C%, is multiplied by eight in the line 70 because the columns take up eight bytes each. All this is to get around the problems caused by the complex way in which the screen is laid out. If we want to be able to move the sprite *up* and *down* a pixel at a time we are going to have to find a way of knowing when to add 640 to the address to get us to the next text line.

For our final sprite routine, we are going to assume that the data for the shape of the sprite is already stored in a section of memory (perhaps in an array) in the format we used in the BASIC example. We can then copy this sprite table into the screen memory at whatever screen position we want the sprite.

Before we can write a complete sprite routine, we should write an experimental version in BASIC. The easiest way to do this is as a procedure.

```
1000 DEF PROCsprite(L%,X%,Y%,W%,H%)
```

Here L% is the location of the first byte of the sprite shape table, X% and Y% are, respectively, the X and Y coordinates of the top left-hand corner of where we want the sprite to appear on the screen, and W% and H% are the width and height of the sprite. To make the program simpler, we will take the X coordinate and the width W% as being in bytes (i.e. they give the number of pixels divided by two). Thus the X coordinate can take values from 0 to 79. The Y coordinate and height H% will be in pixels.

First we need to find the screen memory address of the top left-hand corner of where we want the sprite to appear. We need to split the Y coordinate up into the text row number (from 0 to 31) and the number of pixels down within that row (from 0 to 7). Because each row contains eight pixels vertically, bits 0 to 2 of the Y coordinate will be the number of pixels down within the row and the other five bits will be eight times the row number. So, if we take $Y\% \text{ DIV } 8$, this will give us the text row number. As one row takes up 640 bytes of memory, we must multiply this by 640 then add &3000 (which is the address of the start of the screen). Then we add eight times the X coordinate and finally the least significant three bits of the Y coordinate ($Y\% \text{ MOD } 8$). This will give us the address of the first byte of the screen memory that we will need to change to place the sprite on the screen. We will, however, need to keep the $Y\% \text{ MOD } 8$ part of the address separate as it will tell us how far down within the text row we are. So we end up with something like this:

```
1010 A%=&3000+(Y%DIV8)*640+X%*8
1020 Y%=Y%MOD8
```

Notice that in line 1020 we have altered Y% so that, instead of containing the complete Y coordinate, it now only holds the Y coordinate within the text row.

We now have all the information we need to fix the address of the top left-hand corner of the sprite's intended position on the screen and so place the first row of pixels of the sprite on the screen. By adding eight to the current address (in

A% + Y%) each time, we will move two pixels (= one byte in Mode 2) to the right. We can continue to copy the table into every eighth byte of the screen memory, moving one byte through the table each time, until we have copied the number of bytes specified by the width W%. We won't need the X coordinate again so we can use X% to count bytes.

```
1030 FOR X% = 0 TO W%-1
1040 ?(A%+Y%+X%*8)=?L%
1050 L%=L%+1
1060 NEXT
```

We have now placed the first row of pixels of the sprite on the screen. Now we need to go to the next row of pixels. We can do this by adding 1 to Y%. However, if Y% then equals eight, we need to move to the next text row by adding 640 to A%. At the same time we can check to see if the address has passed through &8000. If so, the sprite has dropped off the bottom of the screen and we want the remainder of it to appear at the top of the screen to produce a 'wrap around' effect. To do this we need only subtract &5000 from A% to move back to the corresponding position at the top of the screen.

```
1070 Y%=Y%+1
1080 IF Y%=8 THEN Y%=0:A%=A%+640:IF
      A%>&7FFF THENA%=A%-&5000
```

Next we need to check whether we have finished drawing the sprite. The easiest way to do this is to subtract one from the height (H%) after each row until H% is 0.

```
1090 H%=H%-1
1100 IF H%>0 THEN 1030
1110 ENDPROC
```

We now have a complete BASIC sprite routine.

However, this routine would be very awkward for realistic animation as it provides no means of removing the sprite again. To do this, we need to

Exclusive-Or the sprite with the screen to put it on and then do the same again to remove it. This is the same technique that can be used from BASIC with the VDU23 user-defined characters. For this we need to alter our program. Line 1040 should now read:

```
1040 ?(A%+Y%+X%*8)=(A%+Y%+X%*8) EOR ?L%
```

Here again, then, is the complete BASIC sprite routine.

```
1000 DEF PROCsprite(L%,X%,Y%,W%,H%)
1010 A%=&3000+(Y%DIV8)*640+X%*8
1020 Y%=Y%MOD8
1030 FOR X% = 0 TO W%-1
1040 ?(A%+Y%+X%*8)=(A%+Y%+X%*8) EOR ?L%
1050 L%=L%+1
1060 NEXT
1070 Y%=Y%+1
1080 IF Y%=8 THEN Y%=0:A%=A%+640:IF
      A%>&7FFF THENA%=A%-&5000
1090 H%=H%-1
1100 IF H%>0 THEN 1030
1110 ENDPROC
```

You may have noticed that this BASIC program is a little awkwardly written. Even so, it has been specifically written to be easy to code into machine code.

At this stage in our exploration it is hardly necessary to give the corresponding line numbers of the sections discussed. You may find it useful to refer to the above listing for the next section.

A machine code sprite routine

For our machine code version we must use a subroutine instead of a procedure. We need a way to pass the parameters (such as the location of the table, X and Y coordinates, etcetera) to the routine. We can use the X and Y registers and some zero page locations to hold these parameters. On entry to the subroutine let's specify that the X and Y registers should initially contain the X and Y coordinates

at which we want to place the sprite; that &72 should contain the width (W%) of the sprite; &73 should contain the height (H%) of the sprite; and that &75 and &76 should contain the start address of the area of memory where we have stored the sprite shape table.

The first thing we need to do is to calculate the screen memory address (A% in our BASIC example). In the BASIC example this was calculated like this:

```
1010 A%=&3000+(Y%DIV8)*640+X%*8
```

Let's deal with this in reverse order. The first task, then, is to calculate $X\%*8$. The answer to this could be larger than 255 (since X% can go up to 79) so we will need two bytes to hold the answer. We are going to store the final answer to A% at &70 and &71 so let's start by putting X% in &70 and zero in &71. Thus we can treat X% as a two-byte number and multiply it by two, three times, so as to finally multiply it by eight. X% is initially contained in the X register, so the code for the multiplication by eight looks like this:

```
STX &70
LDX #0
STX &71
ASL &70
ROL &71
ASL &70
ROL &71
ASL &70
ROL &71
```

Notice that as we don't need the X coordinate again the X register is free for other use.

We have now calculated the $X\%*8$ part of A%, so our next task is the multiplication by 640. Conveniently, the 1.2 Operating System ROM contains a 'times 640' table. This is stored starting at address &C375. It is stored as 32 entries each two bytes long (high byte then low byte). So the contents of &C375 and &C376 are 0 (for 640×0), the contents of &C377

and &C378 are 640 (for 640×1) and so on. (If you don't have a 1.2 Operating System then it is relatively easy to write a BASIC program that calculates such a table and stores it in an array.) So, to add the result of $(Y\%DIV8)*640$ to A% (stored at &70 and &71) we do this:

```
TYA
AND #&F8
LSR A
LSR A
TAX
LDA &C376,X
CLC
ADC &70
STA &70
LDA &C375,X
ADC &71
CLC
ADC #&30
STA &71
```

The first section masks off the bottom three bits of Y% then divides by four. This leaves the accumulator containing the equivalent of $(Y\%DIV8)*2$. This is because each byte in the times 640 table takes up *two* bytes. Thus the result of $(Y\%DIV8)*640$ can be looked up using the X register as a pointer and added to the rest of A% in &70 and &71. Notice that, in the last three commands, we have added &30 to the high byte. This is the equivalent of adding &3000 to the whole number. So now we have the equivalent of A% in &70 and &71. Notice that this is only the address in the screen memory of the first character position at which the sprite is to be placed, not the actual byte within that character position.

The next line in our BASIC example was:

```
1020 Y%=Y%MOD8
```

This represents how far down the character position the sprite is to be placed. Converting this to

machine code is very easy. We want only the least significant three bits of the Y register, so we do:

```
TYA
AND #7
STA &74
```

We have copied this answer into &74 as we are going to need to use it again. The next section of the BASIC program was a loop from 0 to W%-1 for the width of the sprite. Notice that in line 1040 we have to add to A% (now in &70 and &71) the current contents of &74 (Y%) and X%*8. The easiest way to do this in machine code is to first load the Y register with the contents of &74, then add eight to it each time around the loop. If we do this, then the values the loop takes do not matter so long as it is executed the correct number of times.

```
1040 ?(A%+Y%+X%*8)=?(A%+Y%+X%*8) EOR ?L%
```

So, we can load the X register with the width (W%) and decrement it each time round the loop until it is zero. Because we have A% in &70 and &71 and the rest of the expression in the Y register, we can use post-indexed indirect addressing to access the screen, i.e. LDA (&70),Y will be the equivalent of $?(A\% + Y\% + X\% * 8)$.

Next we need to load a byte from the sprite shape table into the accumulator. Because at this stage we are using both the X and Y registers, we cannot easily use any form of indexed addressing. What we do instead is use absolute addressing. In this addressing mode the address is stored as two bytes after the command byte. The first byte is the low byte, the second is the high byte. By changing these two bytes, we can use this addressing mode for looking into tables. This method is not, strictly speaking, 'legal' as it would not work if the routine were in ROM; but it does save on memory and speed.

The next section of our program will look like:

```
row    LDY &74
        LDX &72
```

This sets the Y register to the position of the first pixel of the sprite row and puts the width $W\%$ in the X register. Because we jump back to here at the start of each new row of the sprite, we need the label *row*.

```
.byte LDA &FFFF
```

Above is the command that looks into the sprite shape table; the two bytes after it will be modified by the program as it runs, so the number &FFFF is unimportant. After each byte of the sprite is placed on the screen we will need to jump back to this command; so again a label, *byte*, is needed. However, before we can carry on we need to backtrack to the very beginning of the routine because we need to place the address of the first byte of the sprite shape table at *byte+1* and *byte+2* (low, high) in place of the 'dummy' address we have specified in the assembly code. Thus the program will start by copying the first byte of the table into the screen memory.

We have specified that when calling this routine the address of the first byte of the table should be stored at &75 and &76, so the first four lines of the program need to be:

```
.sprite LDA &75
        STA byte+1
        LDA &76
        STA byte+2
```

Going back to where we left off, we have just loaded a byte from the sprite shape table. So now we will need to EOR this byte with the relevant byte of the screen memory and place the result back on the screen:

```
EOR (&70),Y
STA (&70),Y
```

Then we must add eight to the Y register to move one byte (two pixels) to the right:

```
TYA
CLC
ADC #8
TAY
```

The next line of the BASIC program was:

```
1050 L%=L%+1
```

So we need to add one to the sprite shape table pointer. This is held, remember, in the two bytes after the LDA command at *byte* so we can do this:

```
INC byte+1
BNE nocarry
INC byte+2
.nocarry DEX
BNE byte
```

Notice that we then carry on doing one row of the sprite, moving from left to right, until X has reached zero and we have completed a row.

The next two lines of the BASIC program were:

```
1070 Y%=Y%+1
1080 IF Y%=8 THEN Y%=0:A%=A%+640:IF
A%>&7FFF THEN A%=A%-&5000
```

See if you can spot how this relates to the following machine code.

```
INC &74
LDA &74
CMP #8
BNE notline
LDA #0
STA &74
LDA &70
CLC
ADC #&80
STA &70
LDA &71
ADC #2
STA &71
```

```

        CMP #&80
        BCC notline
        SEC
        SBC #&50
        STA &71
    .notline ...

```

Lastly we need to subtract one from H% and branch back to the label *row* if H% is larger than zero, or return from the subroutine, with the sprite completed, if not.

```

    .notline DEC &73
           BNE row
           RTS

```

We now have a complete assembly language sprite drawing routine.

Listing 1

```

.sprite LDA &75          DEFPROCsprite
                               (L%,X%,Y%,W%,H%)
        STA byte+1
        LDA &76
        STA byte+2
        STX &70          A% =
        LDX #0
        STX &71
        ASL &70          X% * 8
        ROL &71
        ASL &70
        ROL &71
        ASL &70
        ROL &71
        TYA              + (Y%DIV8)
        AND #&F8
        LSR A
        LSR A
        TAX
        LDA &C376,X      * 640
        CLC
        ADC &70
        STA &70
        LDA &C375,X
        ADC &71

```

```

CLC
ADC #&30      + &3000
STA &71
TYA
AND #7        Y%=Y%MOD8
STA &74
LDY &74      .row
LDX &72      FOR X% = 0 TO W%-1
LDA &FFFF    .byte      ?(A%+Y%+X%*8)=
              ?(A%+Y%+X%*8) EOR ?L%
EOR (&70),Y
STA (&70),Y
TYA
CLC
ADC #8
TAY
INC byte+1    L%=L%+1
BNE nocarry
INC byte+2
.nocarry DEX  NEXT
BNE byte
INC &74      Y%=Y%+1
LDA &74      IF Y%<>8
CMP #8
BNE notline  THEN 'notline'
LDA #0       Y%=0
STA &74
LDA &70      A%=A%+640
CLC
ADC #&80
STA &70
LDA &71
ADC #2
STA &71
CMP #&80     IF A%<&8000
BCC notline  THEN 'notline'
SEC         ELSE A%=A%-&5000
SBC #&50
STA &71
.nonline DEC &73  H%=H%-1
BNE row     IF H%>0 THEN 'row'
RTS        ENDPROC

```

This routine will only Exclusive-Or a sprite with the

screen memory; it will not move it for us.

Moving sprites

To move a sprite we have to use the routine twice: first to remove the old image and then to replace it with the new image. It would make the routine easier to use if it would do all this for us. Another point is that, to make a man walk, for example, we would have to replace the previous image with a different image. Ideally our routine should be able to handle this also. Further, we should not have to tell the routine the dimensions and location of the shape table for each sprite every time we want to use it.

The last problem is solved relatively easily by assigning each sprite shape table an arbitrary number by which we can refer to it. First we reserve a section of memory in which to place a table, distinct from the sprite shape table itself, containing all the information on each sprite—an information table. The easiest way to do this is with a DIM statement— *DIM sprites 255* . We will need four bytes of information for each sprite—two bytes for the *address* of its shape table; where the data for the actual sprite shape is stored, and one byte each for *height* and *width* of the sprite—so this command will reserve a table large enough for the information on 64 sprites. If you are not going to use this many sprites, the information table can be smaller. (Notice that we are using one table to point to a series of other tables. This is a very useful technique.)

For convenience let's assume that each sprite takes up four consecutive bytes in this information table: its two-byte shape table address (low-high) followed by its width and then its height (two separate bytes). We can now assign each sprite a number from 0 to 63 where sprite 0 is the one whose data is in the first four bytes of the information table, sprite 1 is the one whose data is in the next four bytes of the information table, and so on.

Now we can enter our new, improved sprite drawing routine with just three parameters—the sprite's X and Y coordinates on the screen in the X

and Y registers, and the sprite's number in the accumulator. Therefore we must now alter the beginning of our sprite routine to handle this.

We will need the X register to point into the sprite information table so we need to save the current contents of the X register. We might as well save this as a two-byte number at &70 and &71 ready for the multiplication by 8. So the modified beginning of the sprite routine starts like this:

```
.sprite STX &70
        LDX #0
        STX &71
```

Next we need to use the sprite number (which we have said must be in the accumulator when the routine is called) as a pointer for the information table. As each entry takes up four bytes we need to multiply the accumulator by four then transfer it to the X register ready to point into the information table.

```
ASL A
ASL A
TAX
```

Now we can transfer the four bytes of information about the sprite (sprite shape table address, width and height) from the information table into the relevant locations ready for the rest of the program.

```
LDA sprites,X
STA byte+1
LDA sprites+1,X
STA byte+2
LDA sprites+2,X
STA &72
LDA sprites+3,X
STA &73
```

Notice that, by taking the address of the sprite shape table directly from the sprite information table and placing it in *byte* we don't need to use locations &75 and &76 any more.

The rest of the routine is the same as before (List-

ing 1 on page 228), starting with:

ASL &70
ROL &71
ASL &70
...
...

We now only have to give the number of the sprite we want to place on the screen, but we still have to move the sprites by putting them on the screen and then taking them off again. This problem can be solved by keeping track of which of the sprites are being shown on the screen at any one time. The best way to do this is to have a number assigned to each moving sprite (or 'film') on the screen. This number should not be confused with the number we assigned to each sprite shape table. If, for example, we had a game with a user-controlled man and eight monsters, the man could be film 0 and the monsters could be films 1 to 8. Any of these nine films (0 to 8) could actually appear as any of the sprites we have shape tables for. And the man would probably alternate between two different sprite shapes so that he would appear to walk.

What we will do is keep a further set of tables that contain the number of the sprite shape that was last used for each film and where on the screen it was placed. To do this we will have three tables.

The table *olds* will contain the number of the last sprite used and *oldx* and *oldy* will contain the last X and Y coordinates for each of the films. Each film will use one byte of each of these three tables. Thus the details for film 0 will be at the first byte of each table, the details for film 1 will be at the second byte of each table, etcetera. So at the beginning of our program we will need:

```
DIM olds 255, oldx 255, oldy 255
```

If fewer films are needed then the size of the arrays can be reduced accordingly. So that we know that none of the films are in use at the start of the program, let's set all the entries in *olds* to 255. So a

255 in *olds* tells the program that the relevant film is not yet active. This means that we can't have a sprite shape numbered 255.

```
FORA%=0T0255
olds?A%=255
NEXT
```

We can now write a new routine, which will use the subroutine *sprite*, that will move a film on the screen. We can enter this routine with just the number of the film we want to move, the number of the new sprite shape we want to use for the film, and the X and Y coordinates to which we want to move the film. For convenience let's keep the sprite number in the accumulator, the X and Y coordinates in the X and Y registers, and store the film number in &75. The first job the new routine must do is to save the contents of the three registers as they will not be needed immediately. We could push these on the stack but this is slow. It is quicker to store them in zero page:

```
.move    STA &76
         STX &77
         STY &78
```

Next we need to load the X register with the film number so that we can look into the film tables. We then need to check if this is a new film (whether the relevant entry in *olds* contains 255). If so, we don't need to remove an old image and can go straight to the new image routine.

```
LDX &75
LDA olds,X
CMP #255
BEQ newfilm
```

If we find that we need to remove an old image we can load the data about the old image from the film tables. Note that because of a shortage of registers we have to temporarily save the contents of the accumulator (which contains the byte from *olds*) in yet

another zero page location.

```
STA &79
LDA oldy,X
TAY
LDA oldx,X
TAX
LDA &79
```

We can now call *sprite* to remove the old image. Notice that we then need to reload the X register with the current film number, as this has been lost.

```
JSR sprite
LDX &75
```

We now have to place the new sprite on the screen and at the same time place the data on the new sprite back in the film tables ready for the next animation. Note that the new sprite shape number and the new X and Y coordinates have to be reloaded from zero page where we stored them temporarily.

```
.newfilm LDA &78
          STA oldy,X
          TAY
          LDA &77
          STA oldx,X
          LDA &76
          STA olds,X
          LDX &77
```

Note again the problems caused by the lack of registers.

Next we need to call *sprite* again. In fact, it is better to place the whole film move routine directly before the *sprite* routine so that the next command will be the start of the *sprite* routine. This saves a JSR command and an RTS command (see full listing below).

We now have a complete *sprite* routine. For clarity here is a complete assembler listing of it. To use it

from BASIC use the command *PROCassemble* at the beginning of the program to initialise it and then call *move* with A%, X%, Y% and &75 set correctly, as explained earlier. (Listing two below.)

Listing 2

```

10000 DEFPROCassemble
10010 DIMsprites 255,olds 255,oldx 255,
      oldy 255
10020 FORA%=0TO255
10030 olds?A%=255
10040 NEXT
10050 DIMZ%200
10060 FORpass%=0TO2STEP2
10070 P%=Z%
10080 [OPTpass%
10090 .move STA &76 \ Store info
10100 STX &77 \ on new sprite
10110 STY &78 \ temporarily
10120 LDX &75 \ in zero page
10130 LDA olds,X
10140 CMP #255
10150 BEQ newfilm
10160 STA &79
10170 LDA oldy,X \ Remove old
10180 TAY \ sprite if nec.
10190 LDA oldx,X
10200 TAX
10210 LDA &79
10220 JSR sprite
10230 LDX &75
10240 .newfilm LDA &78 \ Replace
10250 STA oldy,X \ with
10260 TAY \
10270 LDA &77 \ new sprite
10280 STA oldx,X \ and store
10290 LDA &76 \ new sprite in
10300 STA olds,X \ arrays
10310 LDX &77
10320 .sprite STX &70 \ Main sprite
      \ routine.
10330 LDX #0
10340 STX &71
10350 ASL A
10360 ASL A

```

10370	TAX	\ Get info
10380	LDA sprites,X	\ on sprite shape
10390	STA byte+1	\ from info
10400	LDA sprites+1,X	\ table
10410	STA byte+2	
10420	LDA sprites+2,X	
10430	STA &72	
10440	LDA sprites+3,X	
10450	STA &73	\ Calculate screen
10460	ASL &70	\ addr of top
10470	ROL &71	\ LH corner
10480	ASL &70	\ of sprite
10490	ROL &71	
10500	ASL &70	
10510	ROL &71	
10520	TYA	
10530	AND #&F8	
10540	LSR A	
10550	LSR A	
10560	TAX	
10570	LDA &C376,X	
10580	CLC	
10590	ADC &70	
10600	STA &70	
10610	LDA &C375,X	
10620	ADC &71	
10630	CLC	
10640	ADC #&30	
10650	STA &71	
10660	TYA	
10670	AND #7	
10680	STA &74	
10690	.row LDY &74	\ Plot row
10700	LDX &72	\ of pixels
10710	.byte LDA &FFFF	\ Plot 1 pair of
10720	EOR (&70),Y	\ pixels (1 byte)
10730	STA (&70),Y	
10740	TYA	\ move right to
10750	CLC	\ next pair of
10760	ADC #8	\ pixels
10770	TAY	
10780	INC byte+1	\ set shape table
10790	BNE nocarry	\ pointer to

```

10800          INC byte+2  \ next byte
10810 .nocarry DEX        \ next pair of
10820          BNE byte   \ pixels until
                          \ row complete.
10830          INC &74    \ move down to
10840          LDA &74    \ next row of
10850          CMP #8     \ pixels
10860          BNE notline \ If nec. move
10870          LDA #0     \ down to
10880          STA &74    \ next text row
10890          LDA &70
10900          CLC
10910          ADC #&80
10920          STA &70
10930          LDA &71
10940          ADC #2
10950          STA &71
10960          CMP #&80
10970          BCC notline \ If nec. 'wrap'
10980          SEC        \ back to top
10990          SBC #&50  \ of screen
11000          STA &71
11010 .notline DEC &73   \ Move down a row
11020          BNE row   \ until sprite
11030          RTS      \ complete
11040 ]
11050 NEXT
11060 ENDPROC

```

This routine has several limitations. Firstly, it cannot be used from a second processor because it uses direct screen access rather than the official Acorn commands. For a routine to work with the second processor, only the operating system commands must be used for input and output, but this slows down a machine code program considerably. For most purposes it is better to leave this routine in the main processor and call it from the second processor.

Secondly, notice that as the Y register is used to hold 8 times the X coordinate within the sprite, the largest width of sprite the routine will handle is 32 bytes. Vertically there is no limit. Also note that by using a differently coded sprite shape table, this

routine may be used in any 20K mode. However, the number of horizontal positions the sprite may be positioned at will still only be 80.

Now that we have our sprite routine we must see how it can be used. Let's write a routine to animate a small man around the screen under the control of the Z, X, /, and : keys. We will use the man we used for the BASIC example. First we must have the sprite shape table as data.

```
20000 DATA&0C,&00,&00,&0C
20010 DATA&04,&08,&04,&08
20020 DATA&00,&0C,&0C,&00
20030 DATA&00,&09,&06,&00
20040 DATA&01,&03,&03,&02
20050 DATA&03,&03,&03,&03
20060 DATA&17,&2B,&17,&2B
20070 DATA&17,&21,&12,&2B
20080 DATA&17,&21,&12,&2B
20090 DATA&03,&03,&03,&03
20100 DATA&03,&03,&03,&03
20110 DATA&01,&03,&03,&02
20120 DATA&00,&21,&12,&00
20130 DATA&00,&20,&10,&00
20140 DATA&30,&20,&10,&30
20150 DATA&30,&20,&10,&30
```

Our main program must first go into Mode 2 and turn off the cursor then assemble the machine code. Next it must load the sprite shape table into a reserved block of memory (in this case an array) and place the information about the size of the sprite and location of the shape table in the memory, in *sprites*. We are only going to use one film (film zero) so we can set the film number stored at &75 permanently to zero.

```
10 MODE2
20 VDU23,1,0;0;0;0;
30 PROCassemble
40 DIM man% 63
50 FOR A% = 0 TO 63
60 READ man%?A%
70 NEXT
```

```
80 !sprites = man%
90 sprites?2 = 4
100 sprites?3 = 16
110 ?&75 = 0
```

Notice that the command *READ man%?A%* is legal. This is because expressions using the operators *?*, *!* and *\$* are treated as variable names. Next we set the starting position of the man into *X%* and *Y%* and set the sprite shape number in *A%* to 0.

```
120 X% = 0
130 Y% = 0
140 A% = 0
```

Next we must place the man on the screen.

```
150 CALL move
```

Next we need to alter *X%* and *Y%* according to which keys are being pressed.

```
160 IF INKEY-98 AND X%>0 X%=X%-1
170 IF INKEY-67 AND X%<76 X%=X%+1
180 IF INKEY-105 AND Y%<240 Y%=Y%+4
190 IF INKEY-73 AND Y%>0 Y%=Y%-4
```

Notice the checks to prevent our man from wandering off the edge of the screen. Next we need to go back to line 150 to remove his old image and plot his new one.

```
200 GOTO 150
```

If we now add in the main sprite routine assembly code (Listing 2 page 235) the program should work.

The flicker licker

If you try the program just given, however, you will find that although it is very fast it is also very flickery.

To understand why this is we must first look at the way in which a monitor or TV works. A TV set works on the principle of a beam of electrons which are

fired at a screen. The screen is covered with a substance that glows where the beam hits it. The beam obviously cannot be aimed at the whole screen at one time yet we need the whole screen to appear to glow all the time. In fact what happens is that the beam scans the screen in a series of horizontal lines starting at the top and working down. It completes a full screen (or 'frame') every fiftieth of a second. The result is that the eye is fooled into seeing a permanent picture. This means that if part of our computer's copy of a picture changes it does not appear to change on the screen itself until the beam reaches that point on the screen. So the fastest you can animate a computer image without jarring the eye is 50 times a second. This, however, is also fast enough to fool the eye into seeing continuous motion.

So, our program needs to move the man exactly 50 times a second for him to move smoothly. To move the man we are taking him off the screen and then putting him back on again near where he was. This, of course, means that there is a short space of time when the man is not on the screen at all. If this blank period happens to coincide with the beam's scanning that point the man will just disappear from a whole frame. This results in the flicker that our example program suffers from.

We need, then, a method of synchronising our program with the scanning of the beam in the VDU.

When the beam reaches the bottom of the screen it has to move back to the top again ready for the next frame. While this is happening the screen is 'blanked'. This vertical blanking takes about a fifth of the fiftieth-of-a-second frame cycle.

We should update the screen only during this blanking period. The computer sends a signal (called a synchronisation pulse) to the VDU approximately in the middle of this blanking to tell the VDU to move the beam back to the top again. At the same time as this occurs an interrupt is generated by the computer. When the operating system processes this it subtracts one from the contents of location &240. If we wait until the contents of location &240 change we wait until the synchronisation pulse

occurs, and we can then redraw our picture which will be ready to be 'looked at' when the scanning beam reaches it. *FX19 conveniently does this for us. So we can improve our program by inserting the line:

```
195 *FX19
```

If you try this, however, you will find that now there is a region at the top of the screen where the sprite disappears completely. This is because the synchronisation produced by *FX19 occurs roughly in the middle of the blanking period. The time between this and the beam's starting to draw the top of the screen is too small to move the sprite in. If the sprite is at the top of the screen the beam arrives at the sprite position before it has been replaced; so the sprite vanishes.

Unfortunately there is no simple solution to this problem. In many cases it is easiest to ignore flicker completely. In most slow-moving games, such as PACMAN or DONKEY KONG, where the sprites move only every other frame the flicker will not be much of a problem. Some games, however, could be improved considerably by the removal of flicker.

To do this we need some means of synchronising the sprite movement with the beginning of the blanking period rather than the middle as produced by *FX19. We need an interrupt to occur at the very beginning of the blanking period.

To do this we can use *FX19 to start an interrupt timer in one of the VIA's which will count for just long enough for it to create an interrupt when we need it. Before we can get into the details of this, however, we need to solve another problem. If we are using several films they must all move at the same time—during the blanking period, not when the commands for each of them to move is sent. For this reason we will need a buffer, for each film, which contains the new X and Y coordinates and the new sprite shape number for the next move of that film.

The easiest way to do this is to have three tables *newx*, *newy* and *news*. As with the three tables for

the old positions each film uses one byte of each table. The main program can then set these tables to hold the movements required. When the interrupt occurs, the movements can be processed in one go. We may not want to move all the films every 50th of a second, so any one we don't want to move will have 255 in its corresponding byte of *news*. Once each film has been moved its entry in *news* will be set to 255 ready for the *next* move.

We could use the move routine we already have for this and add to it, but it is easier to rewrite it to handle all the sprites. Firstly we will need the actual sprite routine from LISTING 2, starting at the label *sprite*. Next we need to know how many films are in use at any moment. This can be stored at &77. (Remember that the sprite routine only uses locations &70 to &74.) Now we can start.

If the contents of &77 are, say, 7, then we will refer to bytes 0 to 6 in each of the tables, so we need to load the contents of &77 into the X register and decrement them. While we are at it we need to check whether X was 0 and, if so, end the routine. So the start of the routine will look like this:

```
.rts      RTS
.films    LDX &77
.next     DEX
          CPX #255
          BEQ rts
```

Notice that it is more efficient to place the RTS command before the start of the routine as this ensures that even if the rest of the routine is long the branch range won't be exceeded. The label *next* can be jumped to, to process the next film.

Next we need to check whether this film needs moving. Remember we said that static films would have 255 in *news* (the table holding the new sprite shape number for each film).

```
LDA news,X
CMP #255
BEQ next
```

We now check whether there is an old sprite to remove from the screen and, if so, remove it.

```
LDA olds,X
CMP #255
BEQ newfilm
STA &76
LDA oldy,X
TAY
LDA oldx,X
STX &75
TAX
LDA &76
JSR sprite
LDX &75
```

Now we have to place the new sprite on the screen and store the data on it in the relevant bytes of *olds*, *oldx* and *oldy*.

```
.newfilm LDA news,X
          STA olds,X
          STA &76
          LDA newy,X
          STA oldy,X
          TAY
          LDA newx,X
          STA oldx,X
          STX &75
          TAX
          LDA &76
          JSR sprite
```

Finally we have to store 255 in the relevant byte of *news* (so that the program does not repeat this move in the next frame), and go back for the next film.

```
LDX &75
LDA #255
STA news,X
JMP next
```

Notice the similarities between this version of *move*

and the previous one.

Next we need a routine that will place a command into the buffer. For this purpose we will use &78 to hold the current film number; and the accumulator and the X and Y registers to hold the sprite number and X and Y coordinates respectively, as before. Thus, to instruct the sprite routine to move a film, you must place the film number in &78, the number of the sprite shape you want to use in the accumulator, and the new X and Y coordinates in the X and Y registers. Then you must call the routine *move*.

```
.move    STX &76
         LDX &78
         STA news,X
         TYA
         STA newy,X
         LDA &76
         STA newx,X
         RTS
```

Now we have come to the difficult bit. We need to intercept the interrupt which *FX19 uses and use it to start an interrupt timer in the system VIA. We will need an initialisation routine—first, to copy the IRQ1 vector into a spare vector, and second, to alter it to point to our own interrupt routine. While doing this we need to disable the interrupts so that no interrupt can occur when the vector is half changed.

```
.init    SEI
         LDA &204
         STA &230
         LDA &205
         STA &231
         LDA #irq MOD256
         STA &204
         LDA #irq DIV256
         STA &205
```

We also need to disable the TIMER 1 interrupts and the analogue-to-digital converter interrupts in the system VIA. These tend to cause trouble by interrupting at awkward moments. Doing this means

the analogue-to-digital converter interrupts and the centisecond clock will not work. However, if you want a clock, say for a game, it is easy to add into the *irq* routine a section which counts video sync interrupts (which occur 50 times a second) and increments a location every fiftieth interrupt so that it counts in seconds. It also means that though INKEY will work GET will not.

```
LDA #&50
STA &FE4E
CLI
RTS
```

(See chapter 4 for more information on the registers in the VIA.)

Now that we have initialised the interrupts we need a routine to handle them. This routine must appear totally 'transparent' to the operating system—it mustn't interfere with the operating system at all. For this reason it must not change any of the registers (A, X or Y). When the operating system processes an interrupt it stores the accumulator in location &FC and leaves the other two registers as they were when the interrupt occurred. It then jumps to the vector IRQ1.

We also have the problem that if another interrupt occurs while we are processing the current one, the routine will be entered again. This must not disturb the first entry. For these reasons both the contents of location &FC and the X and Y registers must be pushed on the stack. This saves them so that they can be recovered before returning control to the operating system.

```
.irq    LDA &FC
        PHA
        TXA
        PHA
        TYA
        PHA
```

Next we must check that the interrupt that has

occurred is the one we want. We can look at the VIA for the answer to this.

```
LDA #2
BIT &FE4D
BEQ notsync
```

We now have to deal with the occurrence of a video sync interrupt. We must set TIMER 2 in the system VIA to count for just long enough for the scanning beam in the monitor to reach the vertical blanking period, and then produce an interrupt. So that we can make the fine adjustments later, we use the variable T% to set the time. T% can then be tuned by trial and error to get the best flicker-free picture.

```
LDA &FE4B
AND #&DF
STA &FE4B
LDA &FE4E
ORA #&20
STA &FE4E
LDA #T%MOD256
STA &FE48
LDA #T%DIV256
STA &FE49
```

Lastly we can exit from the routine, remembering to replace all the registers as they were. It is *vital* that, when dealing with interrupts, the registers A, X and Y (and any variables or locations in memory that the main program may be using) hold the same values that they did when the interrupt occurred, before the interrupt-servicing routine passes control back to the operating system.

```
.exit   PLA
        TAY
        PLA
        TAX
        PLA
        STA &FC
        JMP (&230)
```

Now we must check to see if a TIMER 2 interrupt has occurred.

```
.notsync LDA #&20  
          BIT &FE40  
          BEQ exit
```

Then, when a TIMER 2 interrupt occurs we must clear the interrupt status in the VIA ready for the next interrupt.

```
          STA &FE40
```

We are now ready to move the films. Moving the films will take most of the blanking period so the synchronisation interrupt will happen right in the middle of moving the films. For this reason we must ensure that the interrupts are enabled while the films are moved. However, we must not have changed the state of the interrupts when we exit the routine. The easiest way to ensure this is to push the processor status register on the stack before changing the interrupt status and then pull it back off again just before we exit the routine.

```
          PHP  
          CLI  
          JSR films  
          PLP  
          JMP exit
```

We now have the complete routine. For simplicity of use we can dimension all the arrays inside the procedure that assembles the machine code. These arrays need not be longer than the maximum number of sprite shape tables and films that we are going to use, so we might as well specify these two numbers as procedure parameters. At the same time we can set the number of films in use (stored at &77) to the maximum number of films. If, at some point, we want to use fewer films than this number, then we can alter the contents of &77 after using the procedure. While we are about it we may as well clear *olds* and *news*, ready for use, by filling them

with 255. We also need to set T% to a suitable value. This value you can experiment with to get the best results.

Here, then, is the complete sprite routine with films and flicker-prevention.

Listing 3

```
10000 DEFPROCassemble(NF%,NS%)
10010 ?&77=NF%
10020 NF%=NF%-1
10030 NS%=NS%*4-1
10040 T%=18000
10050 DIMsprites NS%,olds NF%,oldx NF%,
      oldy NF%,news NF%,newx NF%,newy NF%
10060 FORA%=0TONF%
10070 olds?A%=255
10080 news?A%=255
10090 NEXT
10100 DIMZ%400
10110 FORpass%=0TO2STEP2
10120 P%=Z%
10130 [OPTpass%
10140 .sprite STX &70      \ draw a sprite
10150         LDX #0       \ given sprite
10160         STX &71     \ no. & XY
10170         ASL A        \ coordinates
10180         ASL A
10190         TAX
10200         LDA sprites,X
10210         STA byte+1
10220         LDA sprites+1,X
10230         STA byte+2
10240         LDA sprites+2,X
10250         STA &72
10260         LDA sprites+3,X
10270         STA &73
10280         ASL &70
10290         ROL &71
10300         ASL &70
10310         ROL &71
10320         ASL &70
10330         ROL &71
10340         TYA
10350         AND #&F8
10360         LSR A
```

10370		LSR A
10380		TAX
10390		LDA &C376,X
10400		CLC
10410		ADC &70
10420		STA &70
10430		LDA &C375,X
10440		ADC &71
10450		CLC
10460		ADC #&30
10470		STA &71
10480		TYA
10490		AND #7
10500		STA &74
10510	.row	LDY &74
10520		LDX &72
10530	.byte	LDA &FFFF
10540		EOR (&70),Y
10550		STA (&70),Y
10560		TYA
10570		CLC
10580		ADC #8
10590		TAY
10600		INC byte+1
10610		BNE nocarry
10620		INC byte+2
10630	.nocarry	DEX
10640		BNE byte
10650		INC &74
10660		LDA &74
10670		CMP #8
10680		BNE notline
10690		LDA #0
10700		STA &74
10710		LDA &70
10720		CLC
10730		ADC #&80
10740		STA &70
10750		LDA &71
10760		ADC #2
10770		STA &71
10780		CMP #&80
10790		BCC notline

10800	SEC	
10810	SBC #&50	
10820	STA &71	
10830	.notline DEC &73	
10840	BNE row	
10850	.rts RTS	
10860		
10870	.films LDX &77	\ Move all films
10880	.next DEX	\ currently
10890	CPX #255	\ active
10900	BEQ rts	
10910	LDA news,X	
10920	CMP #255	
10930	BEQ next	
10940	LDA olds,X	
10950	CMP #255	
10960	BEQ newfilm	
10970	STA &76	
10980	LDA oldy,X	
10990	TAY	
11000	LDA oldx,X	
11010	STX &75	
11020	TAX	
11030	LDA &76	
11040	JSR sprite	
11050	LDX &75	
11060	.newfilm LDA news,X	
11070	STA olds,X	
11080	STA &76	
11090	LDA newy,X	
11100	STA oldy,X	
11110	TAY	
11120	LDA newx,X	
11130	STA oldx,X	
11140	STX &75	
11150	TAX	
11160	LDA &76	
11170	JSR sprite	
11180	LDX &75	
11190	LDA #255	
11200	STA news,X	
11210	JMP next	
11220		

```

11230 .move    STX &76      \ Now
11240         LDX &78      \ move a film in
11250         STA news,X    \ next blanking
11260         TYA          \ period given
11270         STA newy,X    \ film no in &78
11280         LDA &76      \ shape in A
11290         STA newx,X    \ coords in X & Y
11300         RTS
11310
11320 .init    SEI          \ Init. all
11330         LDA &204     \ interrupts
11340         STA &230     \ needed for
11350         LDA &205     \ sprite routines
11360         STA &231
11370         LDA #irq MOD256
11380         STA &204
11390         LDA #irq DIV256
11400         STA &205
11410         LDA #&50
11420         STA &FE4E
11430         CLI
11440         RTS
11450
11460 .irq      LDA &FC      \ Handle any
11470         PHA          \ interrupts
11480         TXA          \ that occur
11490         PHA
11500         TYA
11510         PHA
11520         LDA #2
11530         BIT &FE4D
11540         BEQ notsync
11550         LDA &FE4B
11560         AND #&DF
11570         STA &FE4B
11580         LDA &FE4E
11590         ORA #&20
11600         STA &FE4E
11610         LDA #T%MOD256
11620         STA &FE48
11630         LDA #T%DIV256
11640         STA &FE49
11650 .exit    PLA

```

```

11660          TAY
11670          PLA
11680          TAX
11690          PLA
11700          STA &FC
11710          JMP (&230)
11720 .notsync LDA #&20
11730          BIT &FE4D
11740          BEQ exit
11750          STA &FE4D
11760          PHP
11770          CLI
11780          JSR films
11790          PLP
11800          JMP exit
11810 ]
11820 NEXT
11830 ENDPROC

```

Using the mover

Now we can write a program to show how the routine is used. Firstly we will need this sprite routine and the sprite shape table data from the last example. Then we need a program that will move the sprite around the screen under control of the Z, X, and / keys as with the last example program.

First we go into Mode 2 and remove the cursor.

```

10 MODE2
20 VDU23,1,0;0;0;0;

```

Then we need to assemble the machine code. We are only going to use one film and one sprite so these are the parameters we specify.

```

30 PROCassemble(1,1)

```

Now we need to set up the sprite table.

```

40 DIM man% 63
50 FOR A% = 0 TO 63
60 READ man%?A%
70 NEXT

```

Now we have to set up the parameters of this sprite shape (start of sprite shape table, width and height) in the right table.

```
80 !sprites = man%  
90 sprites?2 = 4  
100 sprites?3 = 16
```

We can now initialise the interrupts.

```
110 CALL init
```

Next we need to put the man on the screen. We can set both the film number and the sprite number permanently to 0. And we can set-up the initial X and Y coordinates. Then we can call the sprite routine to place the man on the screen.

```
120 ?&78 = 0  
130 A% = 0  
140 X% = 0  
150 Y% = 0  
160 CALL move
```

Next we want to check to see if any keys are being pressed, and alter the X and Y coordinates accordingly.

```
170 IF INKEY-98 AND X%>0 X%=X%-1  
180 IF INKEY-67 AND X%<76 X%=X%+1  
190 IF INKEY-105 AND Y%<240 Y%=Y%+4  
200 IF INKEY-73 AND Y%>0 Y%=Y%-4
```

Now we can go back and place the film in its new position.

This sprite routine will only move the film every fiftieth of a second. If we try to move it more often than this, some positions will just be ignored and so the sprite will sometimes jump more than one position at a go. This means, of course, that we can move the film as fast or as slowly as we like and the sprite will still appear flicker free. For our purposes we want the sprite to move one jump every fiftieth of a second. To do this we can use *FX19 to

synchronise the BASIC program with the screen.

```
210 *FX19  
220 GOTO 160
```

If you try this program you will find that unlike the last program it is totally flicker-free and smooth.

Try deleting line 210. You will find that although the man moves a lot faster he appears to move jerkily. There is no flicker but the movement is not smooth. In fact the smoothest speeds at which to move the man turn out to be multiples of 50 moves a second. This means that the man moves exactly the same distance between each frame. So if you want to make a film move faster it is better to increase the distance it moves each time rather than to try and make more moves per second.

This program seems to be perfect for our needs. However, if you use this routine to move more than two or three films you will find that the flicker prevention begins to fail. There is no practical solution to this problem that also allows fifty moves a second for all the films over a full screen. The only way to beat it is to use a smaller amount of the screen. You will find that when flicker occurs it usually occurs only at the top or bottom of the screen. If you ensure that your sprites never move into these areas then your problems are solved. Obviously, this is not always practicable; but if, for instance, you have a 'space invaders' type base at the bottom of the screen it would make sense to make this film zero. This is because lower-number films are processed later (film 0 is processed last) and these will be the ones that are still being processed when the scan starts at the top of the screen so causing possible flicker. By ensuring that film 0 never goes anywhere near the top of the screen you can ensure that it will not flicker. If you do have trouble, try altering the value of T% set at the beginning of the procedure as the best value for this will depend on your program. Beyond this the only advice I can give is to experiment. Do not take these routines as perfect—feel free to customise them to your needs.

Anyone for tennis?

Let's now look at an example of how to use our sprite routine in a simple two-player tennis game. We will use the complete flicker-free sprite routine with films, etcetera. We will need two sprites—a ball and a bat; and three films—one for the ball and one for each bat. We will also need an extra section of assembly code specific to the game.

Because of the length of the assembly code the program will have to be split into three sections that chain each other.

The machine code will take up about 700 bytes; so, as we are using Mode 2 we can place this at addresses &2D00 to &2FFF. Unfortunately, not even the assembly code, let alone the BASIC section of the program, will fit in the space between PAGE and &2D00 on a disc machine. This means that to make the program work on both types of machine we will need to *load* the assembly code in at &3000, *assemble* it to &2D00 and then *chain* the BASIC part of the program in to &1900 or &E00, depending on the filing system. Thus the complete program consists of three sections.

The first section simply loads in the assembly code at &3000.

```
10 MODE7
20 PAGE=&3000
30 CHAIN"TENNIS1"
```

Save this 'loader' first as TENNIS.

Now we must write the assembly code. Because we are using the sprite routine for a specific purpose we can simplify it a bit. This is important to remember—*when using a general routine for a specific purpose always try and simplify it so that it only does what is needed of it.*

Because we know how many sprites and films we need we can set-up the seven tables that the routine uses as part of the assembly code. The table *sprites* will need to be eight bytes long to accommodate the two sprites. Each of the other six tables will need to be three bytes long to accommodate the three films. We can also initialise the routine with no films ac-

tive, by setting all the bytes of *olds* and *news* to &FF. All this data can be added to the end of the assembly code.

```
.sprites EQU 0
          EQU 0
.olds    EQUW &FFFF
          EQUW &FF
.oldx    EQUW 0
          EQUW 0
.oldy    EQUW 0
          EQUW 0
.news    EQUW &FFFF
          EQUW &FF
.newx    EQUW 0
          EQUW 0
.newy    EQUW 0
          EQUW 0
```

This simplifies the first few lines of the assembler section of the program to:

```
10 ?&77=3
20 T%=18000
30 FORpass%=0 TO 2 STEP 2
40 P%=&2D00
50 [OPTpass%
60 .sprite STX &70
..
..
```

The assembly code part of the routine is as before. (see LISTING 3).

One problem is that the third section—the BASIC part of the game—will need to refer to some of the labels in the assembly code, but CHAIN clears all normal variables. The answer to this problem is to copy the four labels we are going to need into integer variables A%, B%, C% and D% which are only cleared on CTRL BREAK. Then the first four lines of the BASIC section can copy them back into the normal variables for easy use.

Let's now look at the beginning of the BASIC section of the program and ignore the rest of the as-

sembly code for a moment. As we have said, the first four lines must be:

```
10 sprites=A%
20 init=B%
30 move=C%
40 tennis=D%
```

Note that *tennis* is the extra assembly code routine we still have to write. Before we write this we should look at the main, BASIC, part of the program.

The next thing we need to do is set the PRINT format for printing the score (see page 325 of the *User Guide*).

```
50 @%=3
```

Now we can go into Mode 2 and set HIMEM so as to safeguard the machine code that the second section of the program has placed from &2D00 to &2FFF. We can turn off the cursor and disable the copy cursor. (This otherwise tends to produce annoying results if the cursor keys are pressed while the game is being played.)

```
60 MODE2
70 HIMEM=&2D00
80 VDU23,1,0;0;0;0;
90 *FX4,1
```

The next job is to print the scores at the top of the screen in appropriate colours.

```
100 COLOUR1:COLOUR130
110 PRINT"SCORE:0    SCORE:0 "
```

Now we must draw the tennis court.

```
120 GCOL0,2:MOVE0,0:MOVE1279,0
130 PLOT85,0,12:PLOT85,1279,12
140 MOVE0,988:MOVE1279,988
150 PLOT85,0,976:PLOT85,1279,976
160 GCOL0,3:FORA%=24T0972STEP32
170 MOVE632,A%:DRAW632,A%+12
```

```
180 MOVE640,A%:DRAW640,A%+12
190 NEXT
```

We have two players, so we need two variables to store their scores. The best way to do this is with an array.

```
200 DIMscore(1)
```

Next we must set-up the designs for the two sprites. For this game we only need very simple sprites. The bat can be made of a red rectangle two pixels (one byte) by 24 pixels (24 bytes). The ball will be a cyan square 2 pixels (1 byte) by four pixels (4 bytes). This means that between them the two sprites need 28 bytes of storage. We can reserve these with the DIM command and set-up the sprite with some simple BASIC.

```
210 DIMsp%28
220 FORA%=0TO23:sp%?A%=3:NEXT
230 sp%!24=&3C3C3C3C
```

Next we need to put the data on the sizes, and addresses, of these two sprite shapes in *sprites*.

```
240 !sprites=sp%+&18010000
250 sprites!4=sp%+24+&4010000
```

This makes the bat sprite 0 and the ball sprite 1. We can now initialise the sprite routine (the films have already been turned off in the assembler section).

```
260 CALLinit
```

We are now into the main section of the game. Let's specify some of the zero page locations we will use.

Locations &70 to &78 are used by the sprite routine. Let's say the &7E and &7F give the current X and Y coordinates of the ball. We can then say that &79 contains the current direction of the ball horizontally—if it is 1, then the ball is travelling to the right, if it is 255 the ball is travelling to the left. Location &7A can contain the vertical speed of the

ball. Because we can position the ball more accurately vertically than horizontally we will make the ball travel left and right at a constant speed while travelling up and down at a variable speed. This will give a range of speeds and directions the ball can take. Location &7A will contain the number of pixels moved down for each position moved along. Positive numbers will be moving down, negative numbers will be moving up.

We now need to initialise these ready for a serve. We shall say that the end that the ball appears at is random and that if it appears on the left it will subsequently travel right, and vice versa. So we now have to decide which end the ball starts. We can set the horizontal direction to either 1 or -1 first and use this to set the X coordinate of the ball either to 0 or 79. The Y coordinate will always be 12 so that the ball starts at the top.

```
270 ?&79=RND(2)*2-3
280 ?&7E=(?&79=255)*-79
290 ?&7F=12
```

We can now set the vertical speed of the ball randomly between 0 and 3. Having done this we must place the ball on the screen for the players to see. The ball is film 0.

```
300 ?&7A=RND(4)-1
310 ?&78=0
320 A%=1
330 X%=?&7E
340 Y%=?&7F
350 CALLmove
```

The bats will be at either side of the screen and will only move up and down. We can say that the Y coordinate for the left-hand bat is in &7C and for the right-hand bat is in &7D. We will start both bats half-way up the screen and, again, we need to place them on the screen. The left-hand bat is film 1 and the right-hand bat is film 2.

```
360 ?&7C=124
370 ?&7D=124
```

```
380 ?&7B=1
390 A%=0
400 X%=0
410 Y%=?&7C
420 CALLmove
430 ?&7B=2
440 X%=79
450 CALLmove
```

Now we have to start the game. We are going to do all the movement of the sprites from machine code. We still have to write the routine (*tennis*) which handles all the movement of ball and bats until one of the players misses the ball. When this happens the routine will exit with the number of the player who won that volley (0 for left and 1 for right) in &7B. So, we can call this routine, then update the score, then have a slight pause, and then serve the next ball. This gives us:

```
460 CALLtennis
470 score(?&7B)=score(?&7B)+1
480 PRINTTAB(6,0);score(0);TAB(17,0);
    score(1)
490 FORA%=1TO10000:NEXT
500 GOTO270
```

Now we must go back to the second section of the game and write *tennis*. The first thing this must do is wait for the vertical sync so as to make the routine move the films every fiftieth of a second. Then we must move the three sprites to their current positions.

```
.tennis LDA #19
        JSR &FFF4
        LDA #0
        STA &7B
        LDA #1
        LDX &7E
        LDY &7F
        JSR move
```

This moves the ball.

```
INC &78
LDA #0
LDX #0
LDY &7C
JSR move
INC &78
LDA #0
LDX #79
LDY &7D
JSR move
```

This moves the two bats.

Next, we have to alter &7E and &7F to point to the next position of the ball. To move it horizontally all we have to do is add the horizontal direction and the sign will take care of which way the ball moves. We can do the same sort of thing for the vertical direction.

```
LDA &7E
CLC
ADC &79
STA &7E
LDA &7F
CLC
ADC &7A
STA &7F
```

Next we must check to see if the ball has hit the top wall of the court. If so, then the Y coordinate will be less than 12 so that we want the ball to bounce. First, it mustn't go through the wall, so we must reposition it to be just touching the wall. The horizontal direction must remain the same but the ball must begin to travel in the opposite direction vertically with the same speed as it hit the wall vertically. This means that we want to multiply the vertical speed by -1 . The easiest way to do this is to subtract it from 0.

```
CMP #12
BCS skip1
LDA #12
```

```
STA &7F
LDA #0
SEC
SBC &7A
STA &7A
```

Next we need to check whether the ball has hit the bottom wall of the court. We can deal with this in a similar way.

```
.skip1 LDA &7F
      CMP #249
      BCC skip2
      LDA #248
      STA &7F
      LDA #0
      SEC
      SBC &7A
      STA &7A
```

Next we need to check whether the ball has reached the left-hand side of the screen. If so, we need to check whether it has hit the bat. If the contents of &7F fall outside the range of ?&7C - 4 to ?&7C + 23 then the ball has missed the bat and the right-hand player has won. Otherwise, we must make the ball bounce off the bat.

```
.skip2 LDA &7E
      BNE skip4
      LDA &7F
      CLC
      ADC #4
      CMP &7C
      BCC rwin
      LDA &7C
      CLC
      ADC #23
      CMP &7F
      BCS skip3
.rwin  LDA #1
      STA &7B
      RTS
.skip3
```

To make the ball bounce, we first have to make it travel to the right. We also have to set the vertical speed of the ball to a random value to simulate the range of shots a real player could play. We must, in a moment, write the routine *rand* to do this.

```
.skip3 LDA #1
        STA &79
        JSR rand
.skip4 ...
```

For *rand* we need to call the random number routine from the BASIC ROM. For BASIC I this is at &AFB6 and for BASIC II this is at &AF87. This routine creates a 33-bit random number in the zero page locations &0D to &11. We need a random number between -3 and 4. (The slight bias to the direction is not enough to matter and makes the coding easier.) To get this we need to take a three-bit random number between 0 and 7 and subtract 3. To get a three-bit random number we need only take any three bits of the 33-bit random number that BASIC generates.

```
.rand JSR &AF87
      LDA &D
      AND #7
      SEC
      SBC #3
      STA &7A
      RTS
```

We can now check for the ball reaching the right-hand side of the screen in the same way.

```
.skip4 LDA &7E
      CMP #79
      BNE skip6
      LDA &7F
      CLC
      ADC #4
      CMP &7D
      BCC lwin
      LDA &7D
```

```

                                CLC
                                ADC #23
                                CMP &7F
                                BCS skip5
.lwin    LDA #0
                                STA &7B
                                RTS
.skip5   LDA #255
                                STA &79
                                JSR rand

```

Next we must move the bats. Before we do this we can simplify matters by writing an INKEY routine. We will use the OSBYTE equivalent of INKEY with a negative number to test individual keys. For this we need to set A to &81, Y to &FF and X to the key number. We then call &FFF4 and compare X with &FF. We can do most of this in a subroutine.

```

.key     LDA #&81
                                LDY #&FF
                                JSR &FFF4
                                CPX #&FF
                                RTS

```

If we jump to this with the key number in X, then, on exit, the zero flag will be set if the key is pressed and be clear if it is not.

If the left-hand bat is not already at the bottom of the screen then we can test the Z key. If it is pressed, we move the bat down four pixels.

```

.skip6   LDA &7C
                                CMP #228
                                BEQ skip7
                                LDX #&9E
                                JSR key
                                BNE skip7
                                LDA &7C
                                CLC
                                ADC #4
                                STA &7C

```

We can do the same sort of thing for moving up

under control of the A key, provided that the bat isn't already at the top.

```
.skip7  LDA &7C
        CMP #12
        BEQ skip8
        LDX #&BE
        JSR key
        BNE skip8
        LDA &7C
        SEC
        SBC #4
        STA &7C
```

We can then do the same for the right-hand bat under control of the] and SHIFT keys.

```
.skip8  LDA &7D
        CMP #228
        BEQ skip9
        LDX #&FF
        JSR key
        BNE skip9
        LDA &7D
        CLC
        ADC #4
        STA &7D
.skip9  LDA &7D
        CMP #12
        BEQ skip10
        LDX #&A7
        JSR key
        BNE skip10
        LDA &7D
        SEC
        SBC #4
        STA &7D
```

Finally, we need to go back to the beginning and repeat the whole process over and again until someone misses the ball.

```
.skip10 JMP tennis
```

This, of course, is a very simple game; but it illustrates the techniques used for much more complicated games.

Here is a listing of all three sections of the program. Type in Listing 4 and save it as TENNIS; then type in Listing 5 and save it as TENNIS1; and then type in Listing 6 and save it as TENNIS2.

By this stage you should have become reasonably familiar with assembly code. You might like to try and follow these three listings, as I have purposely omitted the comments, and work out how they work for yourself. If you get lost, refer back to the program descriptions in this chapter.

Listing 4

```
10 MODE7
20 PAGE=&3000
30 CHAIN"TENNIS1"
```

Listing 5

```
10 ?&77=3
20 T%=18000
30 FORpass%=0TO2STEP2
40 P%=&2000
50 [OPTpass%
60 .sprite STX &70
70         LDX #0
80         STX &71
90         ASL A
100        ASL A
110        TAX
120        LDA sprites,X
130        STA byte+1
140        LDA sprites+1,X
150        STA byte+2
160        LDA sprites+2,X
170        STA &72
180        LDA sprites+3,X
190        STA &73
200        ASL &70
210        ROL &71
220        ASL &70
230        ROL &71
240        ASL &70
250        ROL &71
260        TYA
```

```
270      AND #&F8
280      LSR A
290      LSR A
300      TAX
310      LDA &C376,X
320      CLC
330      ADC &70
340      STA &70
350      LDA &C375,X
360      ADC &71
370      CLC
380      ADC #&30
390      STA &71
400      TYA
410      AND #7
420      STA &74
430 .row  LDY &74
440      LDX &72
450 .byte LDA &FFFF
460      EOR (&70),Y
470      STA (&70),Y
480      TYA
490      CLC
500      ADC #8
510      TAY
520      INC byte+1
530      BNE nocarry
540      INC byte+2
550 .nocarry DEX
560      BNE byte
570      INC &74
580      LDA &74
590      CMP #8
600      BNE notline
610      LDA #0
620      STA &74
630      LDA &70
640      CLC
650      ADC #&80
660      STA &70
670      LDA &71
680      ADC #2
690      STA &71
```

700	CMP	#&80
710	BCC	notline
720	SEC	
730	SBC	#&50
740	STA	&71
750	.notline	DEC &73
760	BNE	row
770	.rts	RTS
780	.films	LDX &77
790	.next	DEX
800	CPX	#255
810	BEQ	rts
820	LDA	news,X
830	CMP	#255
840	BEQ	next
850	LDA	olds,X
860	CMP	#255
870	BEQ	newfilm
880	STA	&76
890	LDA	oldy,X
900	TAY	
910	LDA	oldx,X
920	STX	&75
930	TAX	
940	LDA	&76
950	JSR	sprite
960	LDX	&75
970	.newfilm	LDA news,X
980		STA olds,X
990		STA &76
1000		LDA newy,X
1010		STA oldy,X
1020		TAY
1030		LDA newx,X
1040		STA oldx,X
1050		STX &75
1060		TAX
1070		LDA &76
1080		JSR sprite
1090		LDX &75
1100		LDA #255
1110		STA news,X
1120		JMP next

```

1130 .move    STX &76
1140         LDX &78
1150         STA news,X
1160         TYA
1170         STA newy,X
1180         LDA &76
1190         STA newx,X
1200         RTS
1210 .init    SEI
1220         LDA &204
1230         STA &230
1240         LDA &205
1250         STA &231
1260         LDA #irq MOD256
1270         STA &204
1280         LDA #irq DIV256
1290         STA &205
1300         LDA #&50
1310         STA &FE4E
1320         CLI
1330         RTS
1340 .irq     LDA &FC
1350         PHA
1360         TXA
1370         PHA
1380         TYA
1390         PHA
1400         LDA #2
1410         BIT &FE4D
1420         BEQ notsync
1430         LDA &FE4B
1440         AND #&DF
1450         STA &FE4B
1460         LDA &FE4E
1470         ORA #&20
1480         STA &FE4E
1490         LDA #T%MOD256
1500         STA &FE48
1510         LDA #T%DIV256
1520         STA &FE49
1530 .exit    PLA
1540         TAY
1550         PLA

```

1560		TAX
1570		PLA
1580		STA &FC
1590		JMP (&230)
1600	.notsync	LDA #&20
1610		BIT &FE4D
1620		BEQ exit
1630		STA &FE4D
1640		PHP
1650		CLI
1660		JSR films
1670		PLP
1680		JMP exit
1690	.sprites	EQU D 0
1700		EQU D 0
1710	.olds	EQU W &FFFF
1720		EQU B &FF
1730	.oldx	EQU W 0
1740		EQU B 0
1750	.oldy	EQU W 0
1760		EQU B 0
1770	.news	EQU W &FFFF
1780		EQU B &FF
1790	.newx	EQU W 0
1800		EQU B 0
1810	.newy	EQU W 0
1820		EQU B 0
1830	.tennis	LDA #19
1840		JSR &FFF4
1850		LDA #0
1860		STA &78
1870		LDA #1
1880		LDX &7E
1890		LDY &7F
1900		JSR move
1910		INC &78
1920		LDA #0
1930		LDX #0
1940		LDY &7C
1950		JSR move
1960		INC &78
1970		LDA #0
1980		LDX #79

1990		LDY &7D
2000		JSR move
2010		LDA &7E
2020		CLC
2030		ADC &79
2040		STA &7E
2050		LDA &7F
2060		CLC
2070		ADC &7A
2080		STA &7F
2090		CMP #12
2100		BCS skip1
2110		LDA #12
2120		STA &7F
2130		LDA #0
2140		SEC
2150		SBC &7A
2160		STA &7A
2170	.skip1	LDA &7F
2180		CMP #249
2190		BCC skip2
2200		LDA #248
2210		STA &7F
2220		LDA #0
2230		SEC
2240		SBC &7A
2250		STA &7A
2260	.skip2	LDA &7E
2270		BNE skip4
2280		LDA &7F
2290		CLC
2300		ADC #4
2310		CMP &7C
2320		BCC rwin
2330		LDA &7C
2340		CLC
2350		ADC #23
2360		CMP &7F
2370		BCS skip3
2380	.rwin	LDA #1
2390		STA &7B
2400		RTS
2410	.skip3	LDA #1

2420		STA &79
2430		JSR rand
2440	.skip4	LDA &7E
2450		CMP #79
2460		BNE skip6
2470		LDA &7F
2480		CLC
2490		ADC #4
2500		CMP &7D
2510		BCC lwin
2520		LDA &7D
2530		CLC
2540		ADC #23
2550		CMP &7F
2560		BCS skip5
2570	.lwin	LDA #0
2580		STA &7B
2590		RTS
2600	.skip5	LDA #255
2610		STA &79
2620		JSR rand
2630	.skip6	LDA &7C
2640		CMP #228
2650		BEQ skip7
2660		LDX #&9E
2670		JSR key
2680		BNE skip7
2690		LDA &7C
2700		CLC
2710		ADC #4
2720		STA &7C
2730	.skip7	LDA &7C
2740		CMP #12
2750		BEQ skip8
2760		LDX #&BE
2770		JSR key
2780		BNE skip8
2790		LDA &7C
2800		SEC
2810		SBC #4
2820		STA &7C
2830	.skip8	LDA &7D
2840		CMP #228


```

2850      BEQ skip9
2860      LDX #&FF
2870      JSR key
2880      BNE skip9
2890      LDA &7D
2900      CLC
2910      ADC #4
2920      STA &7D
2930      .skip9 LDA &7D
2940      CMP #12
2950      BEQ skip10
2960      LDX #&A7
2970      JSR key
2980      BNE skip10
2990      LDA &7D
3000      SEC
3010      SBC #4
3020      STA &7D
3030      .skip10 JMP tennis
3040      .rand  JSR &AF87
3050      LDA &D
3060      AND #7
3070      SEC
3080      SBC #3
3090      STA &7A
3100      RTS
3110      .key   LDA #&81
3120      LDY #&FF
3130      JSR &FFF4
3140      CPX #&FF
3150      RTS
3160 ]
3170 NEXT
3180 A%=sprites
3190 B%=init
3200 C%=move
3210 D%=tennis
3220 PAGE=&1900
3230 CHAIN"TENNIS2"

```

Listing 6

```

10 sprites=A%
20 init=B%

```

```
30 move=C%
40 tennis=D%
50 @%=3
60 MODE2
70 HIMEM=&2D00
80 VDU23,1,0;0;0;0;
90 *FX4,1
100 COLOUR1:COLOUR130
110 PRINT"SCORE:0 SCORE:0 "
120 GCOL0,2:MOVE0,0:MOVE1279,0
130 PLOT85,0,12:PLOT85,1279,12
140 MOVE0,988:MOVE1279,988
150 PLOT85,0,976:PLOT85,1279,976
160 GCOL0,3:FORA%=24T0972STEP32
170 MOVE632,A%:DRAW632,A%+12
180 MOVE640,A%:DRAW640,A%+12
190 NEXT
200 DIMscore(1)
210 DIMsp%28
220 FORA%=0T023:sp%?A%=3:NEXT
230 sp%!24=&3C3C3C3C
240 !sprites=sp%+&18010000
250 sprites!4=sp%+24+&4010000
260 CALLinit
270 ?&79=RND(2)*2-3
280 ?&7E=(?&79=255)*-79
290 ?&7F=12
300 ?&7A=RND(4)-1
310 ?&78=0
320 A%=1
330 X%=?&7E
340 Y%=?&7F
350 CALLmove
360 ?&7C=124
370 ?&7D=124
380 ?&78=1
390 A%=0
400 X%=0
410 Y%=?&7C
420 CALLmove
430 ?&78=2
440 X%=79
450 CALLmove
```

```
460 CALL tennis
470 score(?&7B)=score(?&7B)+1
480 PRINTTAB(6,0);score(0);TAB(17,0);
    score(1)
490 FORA%=1TO10000:NEXT
500 GOTO270
```

Conclusion

I hope that you know more about assembly language programming than you did when you first opened this book. You should now feel confident to alter the programs to suit your own tastes and possibly to improve them. You will only become completely fluent in assembly language if you sit down and actually create something using it—as in so many crafts, practice makes perfect!

APPENDIX A

Two's-complement table

* -128	10000000	&80	* -64	11000000	&C0	* 0	00000000	&00	* 64	01000000	&40	*
* -127	10000001	&81	* -63	11000001	&C1	* 1	00000001	&01	* 65	01000001	&41	*
* -126	10000010	&82	* -62	11000010	&C2	* 2	00000010	&02	* 66	01000010	&42	*
* -125	10000011	&83	* -61	11000011	&C3	* 3	00000011	&03	* 67	01000011	&43	*
* -124	10000100	&84	* -60	11000100	&C4	* 4	00000100	&04	* 68	01000100	&44	*
* -123	10000101	&85	* -59	11000101	&C5	* 5	00000101	&05	* 69	01000101	&45	*
* -122	10000110	&86	* -58	11000110	&C6	* 6	00000110	&06	* 70	01000110	&46	*
* -121	10000111	&87	* -57	11000111	&C7	* 7	00000111	&07	* 71	01000111	&47	*
* -120	10001000	&88	* -56	11001000	&C8	* 8	00001000	&08	* 72	01001000	&48	*
* -119	10001001	&89	* -55	11001001	&C9	* 9	00001001	&09	* 73	01001001	&49	*
* -118	10001010	&8A	* -54	11001010	&CA	* 10	00001010	&0A	* 74	01001010	&4A	*
* -117	10001011	&8B	* -53	11001011	&CB	* 11	00001011	&0B	* 75	01001011	&4B	*
* -116	10001100	&8C	* -52	11001100	&CC	* 12	00001100	&0C	* 76	01001100	&4C	*
* -115	10001101	&8D	* -51	11001101	&CD	* 13	00001101	&0D	* 77	01001101	&4D	*
* -114	10001110	&8E	* -50	11001110	&CE	* 14	00001110	&0E	* 78	01001110	&4E	*
* -113	10001111	&8F	* -49	11001111	&CF	* 15	00001111	&0F	* 79	01001111	&4F	*
* -112	10010000	&90	* -48	11010000	&D0	* 16	00010000	&10	* 80	01010000	&50	*
* -111	10010001	&91	* -47	11010001	&D1	* 17	00010001	&11	* 81	01010001	&51	*
* -110	10010010	&92	* -46	11010010	&D2	* 18	00010010	&12	* 82	01010010	&52	*
* -109	10010011	&93	* -45	11010011	&D3	* 19	00010011	&13	* 83	01010011	&53	*
* -108	10010100	&94	* -44	11010100	&D4	* 20	00010100	&14	* 84	01010100	&54	*
* -107	10010101	&95	* -43	11010101	&D5	* 21	00010101	&15	* 85	01010101	&55	*
* -106	10010110	&96	* -42	11010110	&D6	* 22	00010110	&16	* 86	01010110	&56	*
* -105	10010111	&97	* -41	11010111	&D7	* 23	00010111	&17	* 87	01010111	&57	*
* -104	10011000	&98	* -40	11011000	&D8	* 24	00011000	&18	* 88	01011000	&58	*
* -103	10011001	&99	* -39	11011001	&D9	* 25	00011001	&19	* 89	01011001	&59	*
* -102	10011010	&9A	* -38	11011010	&DA	* 26	00011010	&1A	* 90	01011010	&5A	*
* -101	10011011	&9B	* -37	11011011	&DB	* 27	00011011	&1B	* 91	01011011	&5B	*
* -100	10011100	&9C	* -36	11011100	&DC	* 28	00011100	&1C	* 92	01011100	&5C	*
* -99	10011101	&9D	* -35	11011101	&DD	* 29	00011101	&1D	* 93	01011101	&5D	*
* -98	10011110	&9E	* -34	11011110	&DE	* 30	00011110	&1E	* 94	01011110	&5E	*
* -97	10011111	&9F	* -33	11011111	&DF	* 31	00011111	&1F	* 95	01011111	&5F	*
* -96	10100000	&A0	* -32	11100000	&E0	* 32	00100000	&20	* 96	01100000	&60	*
* -95	10100001	&A1	* -31	11100001	&E1	* 33	00100001	&21	* 97	01100001	&61	*
* -94	10100010	&A2	* -30	11100010	&E2	* 34	00100010	&22	* 98	01100010	&62	*
* -93	10100011	&A3	* -29	11100011	&E3	* 35	00100011	&23	* 99	01100011	&63	*
* -92	10100100	&A4	* -28	11100100	&E4	* 36	00100100	&24	* 100	01100100	&64	*
* -91	10100101	&A5	* -27	11100101	&E5	* 37	00100101	&25	* 101	01100101	&65	*
* -90	10100110	&A6	* -26	11100110	&E6	* 38	00100110	&26	* 102	01100110	&66	*

* -89 10100111 &A7 * -25 11100111 &E7 * 39 00100111 &27 * 103 01100111 &67 *
 * -88 10101000 &A8 * -24 11101000 &E8 * 40 00101000 &28 * 104 01101000 &68 *
 * -87 10101001 &A9 * -23 11101001 &E9 * 41 00101001 &29 * 105 01101001 &69 *
 * -86 10101010 &AA * -22 11101010 &EA * 42 00101010 &2A * 106 01101010 &6A *
 * -85 10101011 &AB * -21 11101011 &EB * 43 00101011 &2B * 107 01101011 &6B *
 * -84 10101100 &AC * -20 11101100 &EC * 44 00101100 &2C * 108 01101100 &6C *
 * -83 10101101 &AD * -19 11101101 &ED * 45 00101101 &2D * 109 01101101 &6D *
 * -82 10101110 &AE * -18 11101110 &EE * 46 00101110 &2E * 110 01101110 &6E *
 * -81 10101111 &AF * -17 11101111 &EF * 47 00101111 &2F * 111 01101111 &6F *
 * -80 10110000 &B0 * -16 11110000 &F0 * 48 00110000 &30 * 112 01110000 &70 *
 * -79 10110001 &B1 * -15 11110001 &F1 * 49 00110001 &31 * 113 01110001 &71 *
 * -78 10110010 &B2 * -14 11110010 &F2 * 50 00110010 &32 * 114 01110010 &72 *
 * -77 10110011 &B3 * -13 11110011 &F3 * 51 00110011 &33 * 115 01110011 &73 *
 * -76 10110100 &B4 * -12 11110100 &F4 * 52 00110100 &34 * 116 01110100 &74 *
 * -75 10110101 &B5 * -11 11110101 &F5 * 53 00110101 &35 * 117 01110101 &75 *
 * -74 10110110 &B6 * -10 11110110 &F6 * 54 00110110 &36 * 118 01110110 &76 *
 * -73 10110111 &B7 * -9 11110111 &F7 * 55 00110111 &37 * 119 01110111 &77 *
 * -72 10111000 &B8 * -8 11111000 &F8 * 56 00111000 &38 * 120 01111000 &78 *
 * -71 10111001 &B9 * -7 11111001 &F9 * 57 00111001 &39 * 121 01111001 &79 *
 * -70 10111010 &BA * -6 11111010 &FA * 58 00111010 &3A * 122 01111010 &7A *
 * -69 10111011 &BB * -5 11111011 &FB * 59 00111011 &3B * 123 01111011 &7B *
 * -68 10111100 &BC * -4 11111100 &FC * 60 00111100 &3C * 124 01111100 &7C *
 * -67 10111101 &BD * -3 11111101 &FD * 61 00111101 &3D * 125 01111101 &7D *
 * -66 10111110 &BE * -2 11111110 &FE * 62 00111110 &3E * 126 01111110 &7E *
 * -65 10111111 &BF * -1 11111111 &FF * 63 00111111 &3F * 127 01111111 &7F *

APPENDIX B

This gives all the assembler commands on the 6502 processor and the op-codes of each command.

COMMAND	OP-CODE								
		Z	C5		Z	A6		ABS,Y	F9
		(IND,X)	C1		Z,Y	B6			
		(IND),Y	D1		ABS,Y	BE	SEC	IMP	38
		Z,X	D5	LDY	IMM	A0	SED	IMP	F8
		ABS,X	DD		ABS	AC	SEI	IMP	78
		ABS,Y	D9		Z	A4	STA	ABS	8D
		CPX	IMM	E0	Z,X	B4		Z	85
		ABS	EC		ABS,X	BC		(IND,X)	81
		Z	E4	LSR	ABS	4E		(IND),Y	91
		CPY	IMM	C0	Z	46		Z,X	95
		ABS	CC		A	4A		ABS,X	9D
		Z	C4		Z,X	56		ABS,Y	99
		DEC	ABS	CE	ABS,X	5E	STX	ABS	8E
		Z	C6		NOP	IMP	EA	Z	86
		Z,X	D6		ORA	IMM	09	Z,Y	96
		ABS,X	DE		ABS	0D	STY	ABS	8C
		DEX	IMP	CA	Z	05		Z	84
		DEY	IMP	88	(IND,X)	01		Z,X	94
		EOR	IMM	49	(IND),Y	11	TAX	IMP	AA
		ABS	4D		Z,X	15	TAY	IMP	A8
		Z	45		ABS,X	1D	TSX	IMP	BA
		(IND,X)	41		ABS,Y	19	TXA	IMP	8A
		(IND),Y	51	PHA	IMP	48	TXS	IMP	9A
		Z,X	55	PHP	IMP	08	TYA	IMP	98
		ABS,X	5D	PLA	IMP	68			
		ABS,Y	59	PLP	IMP	28			
		INC	ABS	EE	ROL	ABS	2E		
		Z	E6		Z	26			
		Z,X	F6		A	2A			
		ABS,X	FE		Z,X	36			
		INX	IMP	E8	ABS,X	3E	ROR	ABS	6E
		INY	IMP	C8			Z	66	
		JMP	ABS	4C			A	6A	
		IND	6C				Z,X	76	
		JSR	ABS	20			ABS,X	7E	
		LDA	IMM	A9	RTI	IMP	40		
		ABS	AD		RTS	IMP	60		
		Z	A5		SBC	IMM	E9		
		(IND,X)	A1			ABS	ED		
		(IND),Y	B1			Z	E5		
		Z,X	B5			(IND,X)	E1		
		ABS,X	BD			(IND),Y	F1		
		ABS,Y	B9			Z,X	F5		
		LDX	IMM	A2		ABS,X	FD		
		ABS	AE						

APPENDIX C

This gives all the op-codes and the assembler commands they represent.

00 BRK IMP	32 —	64 —	96 STX Z,Y
01 ORA (IND,X)	33 —	65 ADC Z	97 —
02 —	34 —	66 ROR Z	98 TYA IMP
03 —	35 AND Z,X	67 —	99 STA ABS,Y
04 —	36 ROL Z,X	68 PLA IMP	9A TXS IMP
05 ORA Z	37 —	69 ADC IMM	9B —
06 ASL Z	38 SEC IMP	6A ROR A	9C —
07 —	39 AND ABS,Y	6B —	9D STA ABS,X
08 PHP IMP	3A —	6C JMP IND	9E —
09 ORA IMM	3B —	6D ADC ABS	9F —
0A ASL A	3C —	6E ROR ABS	A0 LDY IMM
0B —	3D AND ABS,X	6F —	A1 LDA (IND,X)
0C —	3E ROL ABS,X	70 BVS REL	A2 LDX IMM
0D ORA ABS	3F —	71 ADC (IND),Y	A3 —
0E ASL ABS	40 RTI IMP	72 —	A4 LDY Z
0F —	41 EOR (IND,X)	73 —	A5 LDA Z
10 BPL REL	42 —	74 —	A6 LDX Z
11 ORA (IND),Y	43 —	75 ADC Z,X	A7 —
12 —	44 —	76 ROR Z,X	A8 TAY IMP
13 —	45 EOR Z	77 —	A9 LDA IMM
14 —	46 LSR Z	78 SEI IMP	AA TAX IMP
15 ORA Z,X	47 —	79 ADC ABS,Y	AB —
16 ASL Z,X	48 PHA IMP	7A —	AC LDY ABS
17 —	49 EOR IMM	7B —	AD LDA ABS
18 CLC IMP	4A LSR A	7C —	AE LDX ABS
19 ORA ABS,Y	4B —	7D ADC ABS,X	AF —
1A —	4C JMP ABS	7E ROR ABS,X	B0 BCS REL
1B —	4D EOR ABS	7F —	B1 LDA (IND),Y
1C —	4E LSR ABS	80 —	B2 —
1D ORA ABS,X	4F —	81 STA (IND,X)	B3 —
1E ASL ABS,X	50 BVC REL	82 —	B4 LDY Z,X
1F —	51 EOR (IND),Y	83 —	B5 LDA Z,X
20 JSR ABS	52 —	84 STY Z	B6 LDX Z,Y
21 AND (IND,X)	53 —	85 STA Z	B7 —
22 —	54 —	86 STX Z	B8 CLV IMP
23 —	55 EOR Z,X	87 —	B9 LDA ABS,Y
24 BIT Z	56 LSR Z,X	88 DEY IMP	BA TSX IMP
25 AND Z	57 —	89 —	BB —
26 ROL Z	58 CLI IMP	8A TXA IMP	BC LDY ABS,X
27 —	59 EOR ABS,Y	8B —	BD LDA ABS,X
28 PLP IMP	5A —	8C STY ABS	BE LDX ABS,Y
29 AND IMM	5B —	8D STA ABS	BF —
2A ROL A	5C —	8E STX ABS	C0 CPY IMM
2B —	5D EOR ABS,X	8F —	C1 CMP (IND,X)
2C BIT ABS	5E LSR ABS,X	90 BCC REL	C2 —
2D AND ABS	5F —	91 STA (IND),Y	C3 —
2E ROL ABS	60 RTS IMP	92 —	C4 CPY Z
2F —	61 ADC (IND,X)	93 —	C5 CMP Z
30 BMI REL	62 —	94 STY Z,X	C6 DEC Z
31 AND (IND),Y	63 —	95 STA Z,X	C7 —

C8 INY IMP
C9 CMP IMM
CA DEX IMP
CB —
CC CPY ABS
CD CMP ABS
CE DEC ABS
CF —
D0 BNE REL
D1 CMP (IND),Y
D2 —
D3 —
D4 —
D5 CMP Z,X

D6 DEC Z,X
D7 —
D8 CLD IMP
D9 CMP ABS,Y
DA —
DB —
DC —
DD CMP ABS,X
DE DEC ABS,X
DF —
E0 CPX IMM
E1 SBC (IND,X)
E2 —
E3 —

E4 CPX Z
E5 SBC Z
E6 INC Z
E7 —
E8 INX IMP
E9 SBC IMM
EA NOP IMP
EB —
EC CPX ABS
ED SBC ABS
EE INC ABS
EF —
F0 BEQ REL
F1 SBC (IND),Y

F2 —
F3 —
F4 —
F5 SBC Z,X
F6 INC Z,X
F7 —
F8 SED IMP
F9 SBC ABS,Y
FA —
FB —
FC —
FD SBC ABS,X
FE INC ABS,X
FF —

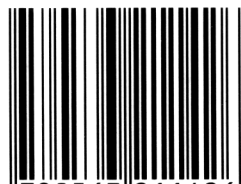
This book reveals the advanced assembly language techniques that are used by professional programmers of arcade and other complex games. Its explanations are at an unsurpassed level of detail, and the substantial routines listed within it may be freely used in your own programs. Major topics covered include assembly code, the operating system, pure machine code, events and interrupts, the keyboard, general graphics, sprites, fill routines, specialised printer screen dumps, and piracy protection. Appendices on the 6502 and 65C02 processor commands complete this thorough examination of a complex but rewarding subject.

If you already have a solid grounding in assembly language, this book will help you to master its heights.

£8.95



ISBN 0-563-21142-3



9 780563 211426