

**CREATIVE ASSEMBLER**  
**How To Write Arcade Games**

**Jonathan Griffiths**

Penguin Books

Creative Assembler How To Write Arcade Games

As one of Acornsoft's games programmers, Jonathan Griffiths was responsible for such widely popular programs as Snapper and JCB Digger for the BBC Microcomputer Model B and Acorn Electron. He has prepared a cassette program, which can be used in conjunction with this book, available from Acornsoft Limited, c/o Vector Marketing Ltd, Denington Industrial Estate, Wellingborough, Northants NN8 2RL.

**Acknowledgements**

Thanks are due to David Johnson-Davies for providing an earlier version of the first few chapters, Jim Dobson and Jeremy Bennett for helping in the writing of this book, and to Philippa Bush and Sharron Fellows for editing it. Also, thanks are due to Orlando M. Pilchard (Q.C.), Chris Jordan, Paul Hudson, Peter Cockerell, Dominic Verity, Jeremy San, Robert Macmillan, Paul Fellows, John Collins, Simon Hughes and Mark Holmes for proof reading.

This book was written and prepared on a BBC Microcomputer Model B using the VIEW word processor.

This book is dedicated to all the staff at Acornsoft.

The Penguin Acorn Computer Library is a joint venture, produced by Acomsoft Limited (in association with Pilot Productions Limited), and published by Penguin Books Limited

Penguin Books Ltd, Harmondsworth, Middlesex, England  
Penguin Books, 40 West 23rd Street, New York, New York, 10010, U.S.A.  
Penguin Books Australia Ltd, Ringwood, Victoria, Australia  
Penguin Books Canada Ltd, 2801 John Street, Markham, Ontario, Canada  
Penguin Books (N.Z.) Ltd, 182-190 Wairau Road, Auckland 10, New Zealand

First published 1984

Copyright © Acomsoft Limited, 1984

All rights reserved

Set in Palatino by Repro Graphics, 61 Cromwell Road, Southampton

Colour origination by RCS Graphics Ltd, 39-40 Springfield Mills, Farsley, Pudsey, Leeds, and MRM Graphics, 61 Station Road, Winslow, Bucks

Made and printed in Spain by Printer industria grafica s.a., Sant Vicenc dels Horts, Barcelona

D.L.B. 15170-1984

Line illustrations by Rob Shone  
Original photography by Nick Wright

Except in the United States of America, this book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's consent in any form of binding or cover other than that in which it is published and without a similar condition including this condition being imposed on the subsequent purchaser

# INTRODUCTION

The BASIC assembler which is available on the BBC Microcomputer and Acorn Electron is a very powerful tool for programmers. It provides a comprehensible interface between the programmer and the machine code language which the 6502 processor itself uses. Hence the programmer is able to control the machine more directly using assembler.

The main reason why people write programs in assembler rather than BASIC is probably because of the speed difference between the two. Assembler instructions can be executed extremely quickly, a program written in BASIC will take between 10 and 100 times as long. Hence assembler is particularly useful for games' programmers since it enables them to move missiles and creatures across the screen quickly and smoothly. If BASIC was used to calculate the new coordinates of each object and draw them at those positions then movement would tend to occur in jerky leaps.

However, speed is not the only factor to be taken into consideration. Assembler programming gives you more power to solve a problem than BASIC does. All high-level languages require programs to have a certain structure and this puts constraints on programs written in that language.

Sceptics may advise against using assembler on the grounds that it is too complex. It is true that operations such as multiplication and division which are easy to perform in BASIC are not as straightforward in assembler. For what might be considered a trivial task, for example multiplying a number by three, several assembler instructions are required instead of just a single BASIC one. A further problem is that there are no FOR ... NEXT or REPEAT ... UNTIL loops in assembler; if you require a loop you must set one up yourself. The same applies to floating point arithmetic - assembler only supports integer calculations.

My advice is that you ignore these sceptics. It isn't difficult to learn to program in assembler. The programs look much less like English than BASIC ones do but nevertheless to someone who knows the language they are easy to understand. Like learning to do anything else, all that is required is a certain amount of knowledge and a lot of practice. This book has been written to provide the knowledge - the rest is up to you.

The book is divided into three sections, each of which has a different task to perform. The first part aims to introduce the more useful assembler instructions available for the 6502 processor, giving simple examples of how they can be used. The second part introduces some of the more complex programming techniques which are aimed in particular at people writing large assembler programs. The third part is aimed mainly at the games' programmer. It provides many useful routines and finally shows how these can all be linked together to produce a complete game.

# 1. DATA STORAGE

Before you can start writing programs in assembler you need to know a few things about how data is stored inside the computer and how that data can be accessed and changed. This chapter looks at the ways in which you can enter numbers from the keyboard and the notation which the computer uses to store these values in its memory.

## 1.1 Hexadecimal notation

To most people it seems natural to use base ten when dealing with numbers. We have ten digits; 0,1,... 8,9, and can use these to represent numbers as large as we please by making the value of a digit depend on which column it is in. Thus, when we consider the number 171 the first '1' represents 100, and the second '1' represents just one. Moving a digit one column to the left multiplies its value by ten; this is why our system is called base 10 or decimal.

When entering numbers into a computer you can still use base 10 if you wish, but another base - base 16 - is also available. For reasons which should become clear as you read through this chapter, base 16 (or hexadecimal) is far more suitable for working with computers. Hence it is advisable at this stage to spend some time becoming familiar with this number system.

In base 16 we need 16 different symbols to represent the 16 different 'hexadecimal digits'. For convenience we retain the symbols 0 to 9, and use the letters A to F to represent the values of ten to fifteen.

Hexadecimal 0 1 2 3 4 5 6 7 8 9 A B C D E F

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

Another difference between base 10 and base 16 is what happens to a digit or hexadecimal digit when it is shifted one column to the left. Whereas we have seen that in base 10 this multiplies the value of the digit by ten, in base 16 it multiplies the value by sixteen. Hence 10 in hexadecimal represents the value sixteen.

Having two bases in which we can work can lead to confusion. Consider, for example, the number 10; as we have seen this can represent either of the values ten or sixteen depending on whether it is being interpreted as a decimal or hexadecimal number. We need a method of specifying whether a number is decimal or hexadecimal. Normally we do this by prefixing hexadecimal numbers with an ampersand (&), e.g.

&B1

The 'B' has the value  $16 \times 11$  because it is in the second column to the left, and the '1' represents 1 unit; the number therefore has the decimal value  $176 + 1 = 177$ .

&123

The '1' is in the third column to the left, so it has the value  $16 \times 16 \times 1$ , the '2' has the value  $16 \times 2$  and the '3' has the value 3. Adding these together produces  $256 + 32 + 3 = 291$

There is no real need to learn how to convert between hexadecimal and decimal because the computer can do it for you, as shown below.

### *Converting hexadecimal to decimal*

To print out the decimal value of a hexadecimal number, such as &123, type

```
PRINT &123
```

The answer, 291, is printed out.

### Converting decimal to hexadecimal

To print, in hexadecimal, the value of a decimal number, type

```
PRINT ~123
```

The answer, 7B, is printed out. The number printed will be in hexadecimal notation, but note that the computer doesn't use the symbol '&' when it is printing hexadecimal numbers. In this case it is obvious that the answer is a hexadecimal number but for an answer such as 79 you would need to know which base you requested the computer to use to be able to interpret the result correctly.

The symbol twiddle or, more accurately, tilde ~ means 'print in hexadecimal'; thus writing

```
PRINT~&123
```

will print 123.

## **1.2 Binary notation and bits**

Although the computer can accept numbers in either decimal or hexadecimal notation, it uses neither of these two systems for storing the numbers in its memory. The computer's memory consists of electronic circuits that can be put into one of two different states. The two states are normally represented as 0 and 1, but they are often referred to by different terms as listed below:

0	1
zero	one
low	high
clear	set
off	on
false	true



### 1.3 Memory locations and bytes

The computer's memory is made up of a number of 'locations', each capable of holding a value. The size of each memory location is normally referred to as a 'byte'. Each byte can hold an eight-bit number, which means that it can store any one of 256 ( $2^8$ ) different values; 0... 255.

We have seen already that each hexadecimal digit requires four bits to specify it. A byte, since it contains eight bits, can therefore represent any hexadecimal number between 0 and &FF.

The bits in a byte are usually numbered for convenience, as follows:

bit number	7	6	5	4	3	2	1	0
byte	0	0	0	1	1	0	0	1

Bit 0 is often referred to as the 'low-order bit' or 'least-significant bit', and bit 7 as the 'high-order bit' or most-significant bit'.

### 1.4 More about memory locations

Somehow it must be possible to distinguish between one location and another. Houses in a town are distinguished by each of them having a unique address. Even when the occupants of a house change, the address of the house remains the same. Similarly, each location in a computer has a unique 'address' consisting of a number which remains unchanged even when the contents of the memory location are altered. Thus we can speak of the 'contents of location 100' as being the number found in the location whose address is 100. The memory locations start from address 0 and could look something like this:

Decimal value of the number being stored:	27	35	6	91
Address:	0	1	2	3

An address can be one or two bytes long. This means that addresses can cover the range 0 to &FFFF. For a detailed look at which part of the memory each address corresponds to see the memory maps in Appendix A.

### Examining memory locations

We can look at the contents of some memory locations in the computer using the query (?) operator. The reference '?X' means use the value of X as the address of the location under consideration. Hence the reference '?&FFEE' means that we are concerned with the location whose address is &FFEE. To look at this location type

```
PRINT ?&FFEE
```

This prints out the value found at the location specified, which in this case should be the number 108. Any memory location can be examined in this way and all of them will contain a number between 0 and 255.



It is often convenient to look at several memory locations in a row; for example, to list the contents of the 32 memory locations from &70 upwards, type  
FOR N = 0 TO 31 : PRINT ?(N+&70); : NEXT N

An alternative way of writing this is

```
FOR N=0 TO 31 : PRINT N?&70; : NEXT N
```

This method is tidier than the other and gives identical results; i.e. for each of the values of N between 0 and 31, N is added to the number &70 to give the address of the location whose contents are to be printed out. This should result in the contents of 32 memory locations being listed on the screen.

### *Changing memory locations*

It is possible to change the number stored at a particular memory location by assigning a new value to it. As an example try changing the contents of &70. First, print the contents of this address; the value there will be whatever was in the memory when the computer was switched on since the computer does not use this location for storing any of the variables it is working with. To change the contents to 7, type

```
?&70=7
```

To verify the change, type

```
PRINT ?&70
```

Try setting the contents of this memory location to other numbers. Setting the contents to a number greater than 255 or &FF will result in the number entered modulo 256 being stored there, for example

```
?&70=600  
PRINT ?&70
```

This will print out

```
(600 MOD 256)
```

A word of warning: Before you change the contents of any other memory locations be sure that you know what you are doing. Although it is quite safe to look at almost any memory location in the computer, care must be exercised when changing any of them. The example given here uses a specific location which is not used by the computer; if you change any other location you may lose any program you have in memory or confuse the computer to such an extent that it proves necessary to reset it by pressing BREAK to make it accept any further commands.



you will notice that there is a certain regularity. The same is true for a b c ~... and the sequence 1 2 3 4

A full table of the ASCII codes used to represent all the characters is given in Appendix A.

## 2. CARRYING OUT INSTRUCTIONS

The previous chapter showed how characters and numbers are represented in the computer's memory, i.e. how the computer deals with data storage. However, data on its own is useless to a computer, it also needs instructions telling it what to do with the data. This chapter looks at how the computer handles instructions and explains some of the simpler assembler instructions which it can use. The instructions listed refer only to the 6502 processor so if you have any other sort of processor connected to your machine, e.g. a Z-80 second processor, this will have to be disabled before attempting any of the routines given in this and subsequent chapters.

### 2.1 The CPU, the computer's brain.

The Central Processing Unit or CPU is the computer's brain. It is the most active part of the computer; although areas of memory can remain unchanged for hours on end when a computer is being used, the CPU is working all the time the machine is switched on. The CPU's job is to read a sequence of instructions from memory and carry out the operations specified by those instructions.

The instructions which the CPU acts on are just values stored in memory locations. The CPU takes a byte and interprets it as an instruction, e.g. `&18` will be interpreted to mean 'clear carry flag'; this will be explained later in this chapter. It then performs the operation as instructed and goes on to collect the next byte.

The first byte of all instructions is the operation code, or 'op-code'. Some instructions, such as the example above, consist of just the op-code; other instructions require data on which they must operate. These instructions therefore consist of two or three bytes, the first one being the op-code and the other one or two consisting of data. For example the value `&E6` is translated into the instruction 'increase the contents of the memory location with the following one-byte address by one'. Hence the CPU then takes the next byte from the memory and interprets this, not as an instruction, but as the address of the location whose contents are to be incremented. It then adds one to the number stored in that location. Having executed this instruction, the CPU then goes on to the next byte which is taken to represent the next instruction it must perform.

### 2.2 Machine code assembler

The above few paragraphs should have given you the idea that everything the CPU acts upon is a number between 0 and `&FF` (255 decimal); each number being interpreted by the CPU as an instruction or some data which an instruction must use. The list of numbers which are being used are referred to as machine code. It is possible for us to talk to the computer in its own language, i.e. program in machine code, but this would mean that we would have to know which instructions all the op-codes stand for. Programming in assembler alleviates the need for learning all these translations. In assembler each op-code is represented by a three letter mnemonic, e.g. `CLC` is used instead of `&18` to give the instruction 'clear carry'. The computer then converts all the mnemonics into the corresponding op-codes.

This process is carried out by an assembler and hence is known as 'assembling'. The program that the assembler takes as its input is known as the source code, and the machine code output is referred to as the object code.

### 2.3 The accumulator and the carry flag

The accumulator is just a temporary location inside the CPU which plays a part in many of the operations performed by the CPU. For example, to add two numbers together you have to load the first number into the accumulator from the memory, add in the second number from memory, and then store the result somewhere. To do this the following assembler instructions will be needed:

Mnemonic	Description	Symbol
LDA	load accumulator from memory	A=M
STA	store accumulator in memory	M=A
ADC	add memory to accumulator with carry	A=A+M+C
CLC	clear carry	C=0

The carry is needed to allow numbers greater than one byte (255 or &FF) to be generated. When an eight-bit value is added to another eight-bit value the result could be too great to be represented by eight bits, e.g.  $140 + 160 = 300 (>255)$ .

In order to allow for this, the CPU will use the carry as the ninth bit of the accumulator, and thus the carry will contain the extra bit. In the above example, when the numbers 140 and 160 are added together and the result stored in a memory location, this location will contain the value 44 ( $300 \text{ MOD } 256$ ). By using the carry flag you will have a record of whether the result of the addition was actually the value 44 or if it was 300. Hence, to avoid confusion, clear the carry before performing any additions.

### 2.4 Writing an assembler program

Enter the following assembler program:

```
10      DIM P% 100
20[
30      LDA &80
40      CLC
50      ADC &81
60      STA &82
70      RTS
80]
90      END
```

The meaning of each line in this assembler program is as follows:

10 The DIM statement is not an assembler mnemonic; it is a BASIC instruction to tell the assembler where to put the assembled machine code by DIMensioning off an area of memory for it. The DIM statement is followed by a number (not in brackets) and the statement reserves this number of bytes for the machine code which will be generated. As a rough guide to the amount of room needed count the number of assembler instructions used, treble it and reserve at least this number of bytes.

The BASIC variable P% is used by the assembler as a location counter to specify the next free address. Hence the statement sets P% to the lowest address of the reserved block of memory and then as each byte of machine code is generated, P% increases by one byte so that it always points to the next free location.

20 The '[' symbol is an 'assembler delimiter' which has to be used immediately before the first assembler statement to tell the BASIC interpreter that the following statements will be in assembler rather than BASIC.

30 Load the accumulator with the contents of the memory location whose address is &80. (The contents of the memory location are not changed.)

40 Clear the carry flag.

50 Add the contents of location &81 to the accumulator with the carry. (Location &81 is not changed by this operation.)

60 Store the contents of the accumulator to location &82. (The accumulator is not changed by this operation.)

70 The RTS instruction will usually be the last instruction of any program; it causes a return to BASIC from the machine-code program. The mnemonic stands for 'return from subroutine'.

80 The ']' symbol is an assembler delimiter which has to be used after the last assembler instruction to tell the interpreter that the following statements will be in BASIC.

90 The END statement is not an assembler mnemonic; it just denotes the end of the program.

Now type RUN and the assembler program will be assembled; the assembled code being inserted directly in memory at the address specified by P%.

An 'assembler listing' will be printed out to show the machine code the assembler has generated to the left of the corresponding assembler mnemonics:

```

>RUN
OE5D
OE5D  A5  80  LDA  &80
OE5F  18      CLC
OE60  65  81  ADC  &81
OE62  85  82  STA  &82
OE64  60      RTS

```

operand  
mnemonic statement  
instruction data/address  
instruction op code  
location counter statement

The program has been assembled in memory starting at &OE5D, immediately after the program text. This address may be different when you enter the example program if you have inserted extra spaces into the program or if you have filing systems other than cassette in your machine, but that will not affect any other part of the listing. All the numbers in the listing are in hexadecimal; thus &18 is the op-code for the CLC instruction, and &A5 is the op-code for LDA when the number being loaded is not given directly but is obtained by looking in the memory location whose one-byte address is given. Hence this LDA instruction consists of two bytes; the first byte is the op-code, and the second byte is the address; &80 in this case.

Another method of finding out where the machine code is, is to find out where 'TOP' is by typing

```
PRINT ~TOP
```

This value gives the address of the memory location immediately after the program text. Since the machine code follows on straight after the text this address is the one corresponding to the first instruction, &A5. Thus the machine code is stored in memory as follows:

```
A5 80 18 65 81 85 82 60
^
TOP
```

When 'RUN' was typed this assembled the assembler program and put the machine code produced into the computer's memory, however it did not execute the program. The method for doing this is described below.

## 2.5 Executing a machine-code program

To execute the machine-code program at TOP, type

```
CALL TOP
```

Nothing obvious will happen except for the '>' prompt being printed again on the screen. This indicates that the computer has finished executing the program and hence the contents of locations &80 and &81 will have been added together and the

results placed in &82.

You can verify this by setting the contents of &80 and &81 to certain values by typing, for example

```
?&80=7 : ?&81=9
```

If you wish you can also set the contents of &82 to 0. Now type

```
CALL TOP
```

and then look at the contents of &82 by typing

```
PRINT ?&82
```

The result is 16 (in decimal); the computer has just added 7 and 9 and obtained 16.

## 2.6 Adding two-byte numbers

Try executing the program for different numbers in &80 and &81. You might like to try the following:

```
?&80=140 : ?&81=160 : CALL TOP
```

We saw earlier in this chapter that if an addition generates a number greater than 255 then the result stored in the memory location specified will be that number modulo 256. Hence the result in this case will be 44 rather than 300. Here is the calculation in hexadecimal:

160	&A0
140	&8C
300	&12C

Only two hex digits can fit in one byte, so the '1' of &12C is lost, and only the &2C is retained. Luckily the '1' carry is retained for us in the carry flag as was mentioned earlier, though we didn't see then how to use this. The example below shows how the two numbers can be treated as being two-byte numbers and added together using the carry to produce a two-byte number which is the complete answer. This method can be extended to any number of bytes since the carry flag makes it a simple matter to add together two numbers as large as we please. Modify the program already in memory by retyping lines 50 to 120, if you wish (leaving out the comments to the right of the assembler text). Here is the modified program:

```
10 DIM P% 100
20[
30     LDA    &80      Low byte of one number
40     CLC                    Clear carry flag
50     ADC    &82      low byte of other number
60     STA    &84      low byte of result
70     LDA    &81      high byte of one number
```



```

80      ADC   &83      high byte of other number
90      STA   &85      high byte of result
100     RTS
110]
123     END

```

Assemble the program:

```

>RUN
OE6E
    OE6E AS   80   LDA   &80
    0E70 18           CLC
    0E71 65   82   ADC   &82
    0E73 85   84   STA   &84
    0E75 AS   81   LDA   &81
    0E77 65   83   ADC   &83
    0E79 85   85   STA   &85
    OE7B 60           RTS

```

Now set up the two numbers as follows:

```

?&81=&8C : ?&81=&00
?&82=&A0 : ?&83=&00

```

Finally, execute the program by typing

```
CALL TOP
```

and look at the result, printing it in hexadecimal this time for convenience:

```
PRINT ~?&84, ~?&85
```

The low byte of the result is &2C, as was obtained

before using the one-byte addition program, but this time the high byte of the result, &1, has been correctly obtained. The carry generated by the first addition was added into the second addition, giving

$0 + 0 + \text{carry} = 1$

Try some other two-byte additions using the new program.

## 2.7 Subtraction

The subtract instruction is just like the add instruction, except that there is a 'borrow' if the carry flag is zero. Therefore to perform a single-byte subtraction the carry flag should first be set with the SEC instruction.

Mnemonic

SEC	set carry flag	C=1
SBC	subtract memory from	A=A-M-(1-C)

## A with carry

### Example

```
10 DIM P% 100
20[
30     LDA     &80     Low byte of first number
40     SEC
50     SBC     &82     Low byte of other number
60     STA     &84     Low byte of result
70     LDA     &81     Now do high bytes
80     SBC     &83
90     STA     &85
100    RTS          Return
110]
120 END
```

Note that the above program is very similar in structure to the addition example in section 2.6.

### 2.8 Comments

There are two methods of putting comments in assembler programs. The first of these, which is used in previous examples, is to put the comment after an assembler instruction, separated from it by one or more spaces, e.g.

```
60 STA&84 low byte of result
```

Alternatively a statement may start with a backslash (\), in which case the remainder of that statement is ignored, e.g.

```
65 \ Now for the high bytes
```

Note that a colon (:) will end the comment and start a new assembler statement, for example line 60 could be replaced by

```
60 \ Low byte of result : STA &84
```

### 2.9 Printing a character

The computer contains routines for the basic operations of printing a character to the VDU, and reading a character from the keyboard, and these routines can be called from assembler programs.

<i>Name</i>	<i>Address</i>	<i>Function</i>
OSWRCH	&FFEE	Puts character in accumulator to output (VDU)
OSRDCH	&FFE0	Reads from input (keyboard) into accumulator

In each case all the other registers are preserved. The names of these routines are acronyms for 'operating system write character' and 'operating system read character' respectively. These routines are executed with the instruction JSR (jump to subroutine).

A detailed description of how the JSR instruction works will be left until the following chapter.

The following program outputs the contents of memory location &80 as a character to the VDU, using a call to the subroutine OSWRCH:

```
10 DIM P% 100
20 oswrch=&FFEE
30[
40 LDA &80
50 JSR oswrch
60 RTS
70]
80 END
```

The variable 'oswrch' is used for the address of the OSWRCH routine. Assemble the program, and then set the contents of &80 to &21 by typing

```
?&80=&21
```

Then execute the program using

```
CALL TOP
```

and an exclamation mark will be printed out before returning to the computer's prompt character, because &21 is the code for an exclamation mark. An alternative method of setting the contents of location &80 to &21 is therefore

```
?&80=ASC"!"
```

Try executing the program with different values in &80, with values chosen from the table of ASCII values in Appendix A.

## 2.10 Immediate addressing

In the previous example the instruction

```
LDA &80
```

loaded the accumulator from the location whose address is &80, this is known as 'absolute' addressing. The location was then set to contain &21, the code for an exclamation mark. If at the time the program was written it was known that an exclamation mark was to be printed in would be more convenient to specify this in the program as the actual data to be loaded into the accumulator. Fortunately an 'immediate' addressing mode is provided which achieves just this. Change the

instruction to

```
LDA #&21
```

where the '#' (hash) symbol specifies to the assembler that immediate addressing is required. Assemble the program again, and note that the instruction op-code for 'LDA #&21' is &A9, not &A5 as it was previously for the absolute addressing. The op-code of the instruction specifies to the CPU whether the following byte is the actual data loaded, or the address of the location containing the data.

## 2.11 Using addresses

So far when a value has been saved, a numerical address has been used to define where it is to be stored, e.g.

```
STA &80
```

A better method of giving an address is to use a variable name, e.g.

```
STA addr
```

In this case 'addr' must be specified at the beginning of the program, e.g.

```
addr = &80
```

This method is better than the previous one since it makes the program easier to understand, i.e. an address can be given a relevant name, e.g. 'xlowbyte' or 'yhighbyte'. In addition, changing the location of a value becomes easier, since only the initial specification need be altered rather than every occurrence of that value throughout the program.

The locations used must be chosen carefully to avoid corrupting operating system or BASIC workspace. The memory map in Appendix A should help to show which locations can be used in different circumstances. Also there are some locations which are always free when using BASIC; these are &70 to &8F.

### 3. JUMPS, BRANCHES AND LOOPS

When an assembler program has been assembled and is being executed, the address of the next instruction to be executed is kept in a register called the 'program counter'. All the programs met so far have been executed in the order that the instructions were written, so the program counter has just steadily increased until it reached the last instruction. This chapter introduces the jump and branch instructions which can make the program counter jump over instructions or move back to previous ones to execute them again. These instructions make it possible to implement loops and perform different instructions depending on the outcome of previous ones.

#### 3.1 Jumps

##### *Ordinary jumps*

Mnemonic	Description
JMP	jump to instruction whose address is given

The JMP instruction is followed by the address of the instruction to be executed next, e.g.

```
JMP &E48    or    JMP addr
```

Instead of describing the address by a number, we can use a 'label' to indicate to the assembler where we want to go. In the assembler, labels are variables prefixed with a full stop (.).

```
10      oswrch =&FFEE
20      DIM P% 100
30[
40.enter
50      LDA #ASC"*"
60.loop
70      JSR oswrch
80      JMP Loop
90]
100 END
```

When the program is assembled the address corresponding to the label '.loop' will be inserted in the machine code. When the code is executed the value of the program counter will be set to this address and the CPU will collect its next instruction from the location with that address and will continue executing from there.

The label '.enter' at the start of the program has been included so that this label can be called in order to execute the program. This is a better way of executing a program than calling TOP since TOP doesn't always point to the first machine code instruction. This is true for the above example since TOP will point to the

assignment statement 'oswrch=&FFEE'.

The program will output an asterisk (\*), and then jump back to the previous instruction. The program has become stuck in an endless loop! Compare this program with the following BASIC program:

```
10 A
20 VDU A
30 GOTO 20
```

To get out of the BASIC loop you press ESCAPE. This will not automatically halt machine code programs, however. To exit from a machine code loop without losing the program you must press BREAK, and then type 'OLD' to retrieve the original program.

### *Jumps to subroutines*

Mnemonic	Description
JSR	jump to subroutine

Examples of this instruction have been used previously. Like the JMP instruction it is followed by a two-byte address, e.g.

JSR oswrch

In this case the address of the instruction directly following the JSR instruction in the code is noted, and then the value of the program counter is set to the address of 'oswrch'. The CPU will go to this address for its next instruction and start executing the code from there until it meets an RTS. This will set the program counter to the address which was noted earlier so that the CPU can then continue executing the code following the JSR instruction. Subroutines jumped to can either be part of the assembler program or, as in this example, sub-routines which exist in the operating system memory.

## **3.2 The zero and negative flags**

There are several flags in the CPU which can be set or cleared depending on the outcome of certain instructions. The carry flag was introduced in the previous chapter, this is set or cleared as the result of an ADC (add with carry) instruction. Another very useful one is the zero flag, called Z. This is set if the result of the previous operation gave zero, and is cleared otherwise, e.g.

LDA &80

would set the zero flag if the contents of &80 were zero.

Similarly the negative flag, N, is set if the result of the previous operation was negative in two's complement notation, i.e. if the top bit was set, e.g.

LDA &80

would set the negative flag if the number stored in location &80 was greater than 127 (01111111).

The conditions of all the flags are stored in a byte called the status register (P), and each flag is represented by one bit: e.g. the top bit of the status register is set if N=1 and the bottom bit is set if C=1.

### 3.3 Conditional branches

Conditional branches enable the program to act on the outcome of an operation. There are eight different branch instructions, six of which are introduced.

Mnemonic	Description	Status
BEQ	branch if equal to zero	(ie Z=1)
BNE	branch if not equal to zero	(ie Z=0)
BCC	branch if carry clear	(ie C=0)
BCS	branch if carry set	(ie C=1)
BPL	branch if plus	(ie N=0)
BMI	branch if minus	(ie N=1)

The conditional branch instructions test the state of the various condition flags, e.g. the zero flag and negative flag. If the condition is not satisfied then it carries on executing, but if the condition is satisfied then the computer goes to the place indicated by the byte following the branch op-code. This byte is stored as a relative address, thus if you say

BCS notzero

the assembler works out the difference (in bytes) between the current instruction and the place where the label '.notzero' is, and puts this value after the op-code. This means that the value of this byte is used, in conjunction with the address of the current instruction, to tell the CPU where to go next.

Because only a single byte is allowed in this relative addressing mode, the branch instructions can only point to one of 255 nearby bytes. The two's complement representation of numbers is used to give the offset relative to the current address. Branches which point forwards are restricted to 0-127 bytes beyond the current location. The value of the byte following the op-code for these is then 0-127. Branches which point backwards to places at lower addresses in memory require a negative value to be added to the current location. These use the numbers 128-255 to represent the values -128 to -1.

The JMP instruction does not use relative addressing; it is followed by two bytes which specify the absolute address which will be the destination. Hence the branch instruction is shorter than the jump instruction, the jump being three bytes long (op-code and two-byte address) and the branch being two bytes long (op-code and one-byte offset). This difference is automatically looked after by the assembler.

The following simple program will print an

exclamation mark if 'character' contains zero, and a star if it does not. The comments to the right of the assembler statements may be omitted when you enter the program.

```
10      DIM P% 100
20      character=&80
30      oswrch = &FFEE
40[
50.enter
60      LDA      character
70      BEQ      exclamation      If zero print '!'
80      LDA      #ASC"*"          Star
90      JSR      oswrch            Print it
100     RTS
110.exclamation
120     LDA      #ASC"!"          Exclamation mark
130     JSR      oswrch            Print it
140     RTS
150]
160 END
```

Note that the above program can be made shorter, by replacing the instructions

```
JSR oswrch
RTS
```

with the single instruction

```
JMP oswrch
```

Replacing JSR and RTS instructions by a JMP to a subroutine reduces the size of both a source program and the object code it produces, and hence increases execution speed.

Now assemble the program by typing RUN. You should get the message:

```
No such variable at Line 70
```

This is because the assembler processes the mnemonic instructions in the order in which they are listed in the program. Therefore when it encounters 'BEQ exclamation' it has not yet found the label 'exclamation' so it cannot work out the offset which is required in the following byte. This is known as the forward-reference problem, and is easily overcome using the method of two-pass assembly which is explained below.

### 3.4 Two-pass assembly

When a program contains forward references it needs to be assembled twice. During the first pass of the assembler the addresses of all the labels are noted so



that during the second pass the offsets of the branch instructions can be included. And the assembler must be told not to worry when, during the first pass, it comes across errors of the sort indicated above.

This can be done using the OPT statement, an assembler directive which has a single parameter for which the following values are possible:

- OPT 0 No error messages, and no listing
- OPT 1 No error messages, and listing
- OPT 2 Error messages reported, and no listing
- OPT 3 Error messages reported, and listing (Default)

Thus to suppress messages and a listing on the first pass, and to restore them on the second pass, we need to use OPT 0 and OPT 3 respectively. This can be effected by placing the directive inside a FOR NEXT loop, which goes from 0 to 3 in steps of 3. Then

The value of the control variable is used as the parameter of the OPT statement. So, to alter the program which was given above, simply enter these lines:

```
10 DIM code 100
23 FOR pass = 0 TO 3 STEP 3
26 P% = code
30[ OPT pass
145 NEXT pass
```

This time the error message will not be produced and the correct offset will be calculated for the branch instruction.

Note lines 10 and 26, which replace the old 'DIM P% 100' statement. P% must be reset to the starting value each time that the code is assembled.

Now execute the program by typing

CALL enter

and verify that the program behaves as it should for different values in &80.

### 3.5 X and Y registers

The CPU contains two registers, called the X and Y registers, in addition to the accumulator. As with the accumulator, there are instructions to load and store the X and Y registers:

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
LDX	load X register from memory	X=M
LDY	load Y register from memory	Y=M
STX	store X register to memory	M=X
STY	store Y register to memory	M=Y

However, unlike the accumulator, the X and Y registers cannot be used as one of the operands in arithmetic instructions; they have their own special uses which will be outlined later.

The X and Y registers are particularly useful as the control variables in iterative loops, because four special instructions exist which will either increment (add 1 to) or decrement (subtract 1 from) their values.

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
INX	increment X register	X=X+1
INY	increment Y register	Y=Y+1
DEX	decrement X register	X=X-1
DEY	decrement Y register	Y=Y-1

Note that these instructions do not affect the carry flag: incrementing &FF will give &00 without changing the carry bit. The zero and negative flags are, however, affected by these instructions.

### 3.6 Iterative loops

The iterative loop enables the same set of instructions to be executed a fixed number of times, e.g.

```

10 DIM P% 100
20 oswrch = &FFEE
30E
40.enter
50      LDX #8          Initialise X
60      LDA #ASC"*"    Code for star
70. Loop
80      JSR oswrch     Output star
90      DEX           Count it
100     BNE           Loop all done?
110     RTS
120     END

```

Assemble the program by typing RUN. This program prints out a star, decrements the X register, and then branches back if the result after decrementing the X register is not zero. Consider what value X will have on successive trips around the loop and predict how many stars will be printed out; then execute the program with 'CALL enter' and see if your prediction was correct. (If you were wrong, try thinking about the case where X was initially set to 1 instead of 8 in line 50.)

How many stars are printed if you change the instruction on line 50 to 'LDX #0'?

### 3.7 Comparing values

In the previous example the condition  $X=0$  was used to terminate the loop. Sometimes we might want to count up from 0 and terminate on some other specified value. The compare instruction can be used to compare the contents of a register with a value in memory; if the two are the same, the zero flag will be set. If they are not the same, the zero flag will be cleared. The compare instruction also affects the carry flag by setting it to 1 if the register is greater than or equal to the value in memory, and 0 otherwise.

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
CMP	compare accumulator with memory	A-M
CPX	compare X register with memory	X-M
CPY	compare Y register with memory	Y-M

Note that the compare instruction does not affect its two operands, it just changes the flags as a result of the comparison.

The next example again prints eight stars, but this time it uses X as a counter to count upwards from 0 to 8.

```
10 DIM P% 100
20 oswrch=&FFEE
30[
40.enter
50   LDX   #0           Start at zero
60..Loop
70   LDA   #ASC"*"     Code for star
80   JSR   oswrch       Output star
90   INX
100  CPX   #8           ALL done?
110  BNE   Loop        If not then repeat
120  RTS              Else return
130]
140 END
```

In this program X takes the values 0, 1, 2, 3, 4, 5, 6, and 7. The last time around the loop X is incremented to 8, and the loop terminates. Try drawing a flowchart for this program.

### 3.8 Using the control variable

In the previous two examples X was simply used as a counter, and so it made no difference whether we counted up or down. However, it is often useful to use the value of the control variable in the program. For example, we could print out the character in the X register each time around the loop. The order in which we want

the characters would then determine whether we count up or down. We therefore need a way of transferring the value in the X register to the accumulator so that it can be printed out by the OSWRCH routine. One way would be to execute:

```
STX tempaddr
LDA tempaddr
```

where 'tempaddr' is not being used for any other purpose. However, there is a more convenient way, using one of four new instructions:

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
TAX	transfer accumulator to X register	X=A
TAY	transfer accumulator to Y register	Y=A
TXA	transfer X register to accumulator	A=X
TYA	transfer Y register to accumulator	A=Y

Note that the transfer instructions only affect the register being transferred to.

The following example prints out the alphabet by making X cover the range A to Z.

```
10 DIM P% 100
20 oswrch = &FFEE
30[
40.enter
50  LDX #ASC"A"      Start with the Letter A
60.loop
70  TXA              Put it in the accumulator
80  JSR oswrch       Print it
90  INX              Next one
100 CPX #(ASC"Z"+1) Finished ?
110 BNE loop         If so - continue
120 RTS              Else return
130]
140 END
```

All these examples could have used Y as the control variable instead of X in exactly the same way.

### 3.9 Conditional assembly

Assembler source text can contain tests, and assemble different statements depending on the outcome of these tests. This is especially useful where slightly different versions of a program are needed for many different purposes. Rather than creating a different source file for each different version, a single variable can determine the changes using conditional assembly, eg:

```
10 DIM CODE%100
20 char=&70
30 oswrch=&FFEE
40 osrdch=&FFE0
50 bell=7          Beep = VDU 7
60 prompt=ASC":."
70 INPUT"bell",bell$      Input 'bell$'
80 bellflag=INSTR("Yy",bell$)  bellflag' is true if
90 FOR pass=0 TO 3 STEP 3    bell$ = 'y' or 'Y'
100 P%=CODE%
110[ OPTpass
120.enter
130]
140 IF bellflag THEN [OPT pass:LDA #bell :JSR oswrch:]
150[OPT pass
160  LDA #prompt          Load A with ':' prompt
170  JSR oswrch           Print from A
180  JSR osrdch          Read in a character
190  STA char             store it at char and
200  JSR oswrch           print it out
210  RTS
220]
230  NEXTpass
```

When this program is run it asks if you want the computer to beep or not and sets 'bellflag' accordingly. Then when the machine code is executed it inputs a character from the keyboard, beeping if 'bellflag' is set to remind you that an input is required, and prints out a character corresponding to the first key pressed. This character is also saved at the address 'char'.

## 4. LOGICAL OPERATIONS, SHIFTS AND ROTATES

We have seen previously that each byte or memory location is made up of eight bits, each of which can be set to the value 0 or 1. Although the operations we have considered so far have treated the whole byte as the smallest quantity being dealt with, many operations in the computer's instruction set are best considered as operations which act on eight separate bits. Some of these perform such important tasks as changing the case of characters, or multiplying and dividing.

### 4.1 Logical operations

Logical operations are performed between the individual bits of two operands; one of the operands is always the accumulator and the other is a memory location or immediate value. In this section three such operations are introduced; AND, OR and EOR. A truth table is used to give a compact description of each operation. This takes two single bit inputs which, for convenience, we call A and B, and shows the bit which is produced as a result of ANDing, ORing or EORing them together. This is known as Boolean logic after its inventor, George Boole.

#### AND

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
AND	AND accumulator with memory	A=A AND M

Truth table:

A	B	out
0	0	0
0	1	0
1	0	0
1	1	1

The AND operation sets a bit of the result to a 1 only if the corresponding bit of one operand is a 1 AND the corresponding bit of the other operand is a 1; otherwise the bit in the result is a zero, e.g.

	<i>Hexadecimal</i>	<i>Binary</i>
operand 1	A9	10101001
operand 2	E5	11100101
result of AND	A1	10100001

One way of thinking of the AND operation is that one operand acts as a 'mask', and only where there are ones in the mask do the corresponding bits in the other operand 'show through'; otherwise, the bits are zero.

## OR

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
ORA	OR accumulator with memory	A=A OR M

Truth table:

A	B	out
0	0	0
0	1	1
1	0	1
1	1	1

The OR operation sets a bit of the result to a 1 if the corresponding bit of one operand is a 1 OR the corresponding bit of the other operand is a 1, or indeed, if they are both ones; otherwise the bit in the result is zero, e.g.

	hexadecimal	Binary
operand 1	A9	10101001
operand 2	E5	11100101
Result of OR	ED	11101101

## Exclusive OR

<i>Mnemonic</i>	<i>Description</i>	<i>Symbol</i>
EOR	Exclusive-OR accumulator with memory	A=A EOR M

Truth table:

A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

The Exclusive-OR operation is like the OR operation, except that a bit in the result is set to 1 only if the corresponding bit of one operand is a 1, or if the corresponding bit of the other operand is a 1, but not if they are both ones, e.g.

	hexadecimal	Binary
operand 1	A9	10101001
operand 2	E5	11100101
result of EOR	4C	01001100

Another way of thinking of the Exclusive-OR operation is that a bit of the result is 1 if and only if the corresponding bits in the operands are different.

### *Example - converting lower to upper case*

The following example converts all characters entered in lower case to upper case. See Appendix A for the ASCII character set.

```
.loop
    JSR osrdch    Get character
    AND #&DF     Make case bit zero
    JSR oswrch   Print it
    JMP loop     And do it again
```

Try altering this using the 'ORA' instruction to convert all characters to lower case. When you have succeeded in doing this try writing a routine to swap case.

## **4.2 The BIT instruction**

This instruction is available to test whether individual bits of a number are set or not.

<i>Mnemonic</i>	<i>Description</i>
BIT	Compare memory bits with accumulator

The instruction AND's the bits of the accumulator and the memory. The zero and negative flags are set or cleared as a result of this operation; Z=1 if the result was 0 and N = top bit, V = bit 6 of contents of location.

Hence BIT may be used to test any bit of the memory by loading the accumulator with a value containing a 1 in the relevant position and 0's everywhere else. Then the values 0 and 1 for Z show whether the bit was or was not set respectively, e.g.

```
LDA #4          4=00000100
BIT addr
BEQ bit-not-set
```

If addr contained, for example, &43 (01000011) then the branch would occur. If, however, addr contained &44 (01000100) then the branch would not take place. Bit differs from the AND instruction in that it does not corrupt the accumulator.



### 4.3 Rotates and shifts

The rotate and shift operations move the bits in a byte either left or right.

<i>Mnemonic</i>	<i>Description</i>
ASL	arithmetic shift left
ROL	rotate left
LSR	logical shift right
ROR	rotate right

The ASL instruction moves all the bits one place to the left; what was the high-order bit is put into the carry flag, and a zero is put into the low-order bit of the byte. The ROL instruction is identical except that the previous value of the carry flag is put into the low-order bit instead of zero.

The right shift and rotate right instructions work in a similar way except that the bits are shifted to the right.

ASL – Arithmetic shift Left one bit

ROL - Rotate Left one bit

LSR - Logical shift right one bit

*Example - multiplying by two*

The most efficient way to multiply a two-byte number, stored in 'addr' and 'addr+1', by two, is to shift the contents of the two bytes one place to the left. Where 'addr' and 'addr+1' are the addresses of the locations storing the low and high bytes of the number being doubled, use the following two statements:

```
ASL addr  
ROL addr+1
```

This works by using the carry to hold the bit that falls off the end of 'addr', and then using the ROL statement, which puts the carry into the correct place in the high-order byte.

## 5. ADDRESSING MODES

So far we have met two addressing modes. One of these is absolute addressing as in

```
LDA addr
```

which, when executed, loads the accumulator with the contents of the location whose address is 'addr'. The other is immediate addressing as in

```
LDA #&81
```

which, when executed, loads the accumulator with the actual value &81.

However, other addressing modes exist and one of the most important, 'indexed addressing', is introduced here prior to a summary exposition of all the addressing modes available to the 6502 processor.

### 5.1 Indexed addressing

In this addressing mode one of the index registers (X or Y) is added to the address as an offset which gives the precise location for the stored data. For example, we can write:

```
LDA addr, X
```

If X contains zero this instruction will behave just like 'LDA addr'. However, if X contains 1 it will load the accumulator with the contents of 'one location further on from addr'. Since X can contain any value from 0 to 255, the instruction 'LDA addr,X' gives you access to 256 different memory locations. If you are familiar with BASIC's byte vectors you can think of 'addr' as the base of a vector, and of X as containing the subscript, e.g.

```
addr?7 = 12
```

is equivalent to

```
LDA #12  
LDX #7  
STA addr, X
```

### 5.2 String types

Two examples of the use of indexed addressing are given below, both involving strings. There are two string types available for use in BASIC and assembler; ATOM strings and Microsoft strings. An ATOM string is a string of characters terminated by a RETURN character. The name which identifies the string is preceded by a dollar (\$) sign and the strings can be easily set up in BASIC, e.g.

```
$name = "Fred"
```

ATOM strings must have an area of memory set aside for them. This can be done, as in the examples, by using a DIM statement. The characters making up the string are then stored in the location identified by the name of the string. This is very useful as the address of each character is then also known.

A Microsoft string is a string of characters preceded by a byte which gives the length of the string. In this case, the name of the string has a dollar (\$) sign *after* it. It is more flexible than the ATOM string because it can contain RETURN characters. Its disadvantage is that all the characters making up the string are stored in locations chosen by BASIC, hence the addresses of these are not known.

*Example - print inverted-case string*

The following program uses indexed addressing to

print out a string of characters terminated by a carriage return (which is represented in the memory by &D), swapping case as it prints out each character.

```
10 DIM string 256, code 100
20 oswrch = &FFEE
30 FOR pass = 0 TO 3
40 P% = code
50[OPT pass
60.enter
70   LDX #0           Set index to zero
80.loop
90   LDA string,X    Get characters from string
100  CMP #&D        Is it end of string?
110  BEQ return     If so, end
120  EOR #&20       Else invert case bit
130  JSR oswrch     Print it
140  INX           Increment index
150  BNE Loop      If string longer than 256
60..return
170  RTS           then end anyway
180]
190 NEXT pass
200 END
```

Assemble the program by typing RUN, and then try the program by entering:

```
$string = "Test String" : CALL enter
```

*Example - index subroutine*

Another useful operation, easily performed in a machine-code routine, is looking up a character in a string and returning its position in that string. The following subroutine reads in a character, using a call to the OSRDCH read-character routine,

and saves in '? found' the position of the first occurrence of that character in '\$target'. This is exactly the same as the BASIC '?found =INSTR ("ABCDEFGH", GET\$)'.

```
0 REM Index Routine
10 DIM target 25,P% 100
20 osrdch=&FFE0 : $target="ABCDEFGH" : found= &70
30[
40.enter
50 JSR osrdch Get character
60 LDX#(LEN($target)-1) Length of string
70.loop
80 CMP target,X Compare character
90 BEQ match Got a match
100 DEX Try again
110 CPX#255 Until end of string
120 BNE loop
130.match
140 INX The position in the
150 STX found String is stored
160 RTS Return
170]
180 END
```

The routine is entered at '.enter', and as it stands it looks for one of the letters A to H.

### 5.3 Summary of addressing modes

The following sections summarise all addressing modes that are available on the 6502, some of which have been met already.

#### *Immediate addressing*

Use immediate addressing when the data for an instruction is known at the time of writing the program. In this mode the second byte of the instruction contains the actual eight-bit data to be used by the instruction. The '#' symbol denotes an immediate operand.

Examples: LDA #value  
CPY #flag+2

#### *Absolute addressing*

Use absolute addressing when the effective address, to be used by the instruction, is known at the time the program is being written. In this mode the two bytes following the op-code contain the 16-bit effective address to be used by the instruction, the low byte being given first, followed by the high byte.

Example: LDA address

### *Zero page addressing*

Zero page addressing is a subset of absolute addressing. They are similar in that the instruction specifies the effective address to be used; the difference between them is that in absolute addressing the address used can be anywhere, whereas in zero page addressing the address is in zero page, i.e. from &0000 to &00FF. Hence this address is only one byte rather than two. The assembler will automatically produce zero-page instructions.

Examples: JSR Loop  
ASL &9A

### *Indexed addressing*

Indexed addressing is used to access a table of memory locations by specifying them in terms of an offset from a base address. The base address is known at the time that the program is written; the offset, which is provided in one of the index registers, can be calculated by the program.

In all indexed addressing modes one of the eight-bit index registers, X and Y, is used in order to calculate the effective address to be used by the instruction. Five different indexed addressing modes are available, and are listed below.

### *Absolute indexed addressing*

The simplest indexed addressing mode is absolute indexed addressing. In this mode the two bytes following the instruction specify a 16-bit address which is to be added to one of the index registers to form the effective address to be used by the instruction.

Examples: LDA table,X  
LDX palette,Y  
INC score,X

### *Zero,X indexed addressing*

Here, the second byte of the instruction specifies an eight-bit address, which is added to the X-register to give a zero-page address to be used by the instruction.

Note that in the case of the LDX instruction a 'zero,Y' addressing mode is provided instead of the 'zero,X' mode.

Examples: LSR &80,X  
LDX addr,Y (where addr+Y is in zero page)

### *Indirect addressing*

It is sometimes necessary to use an address which is actually computed when the

program runs, rather than being an offset from a base address or a constant address. In this case indirect addressing is used.

Indirect addressing is distinct from direct addressing (i.e. absolute, indexed, etc) in that the address specified after the mnemonic is used to refer to a location where the final address will be found. Thus the machine does not go directly to the address, but instead it goes indirectly, via the address given.

The indirect mode of addressing is available for the JMP instruction. Thus control can be transferred to an address calculated at the time the program is run.

Examples:   JMP (&2800)  
              JMP (addr)

For the dual-operand instructions ADC, AND, CMP, EOR, LDA, ORA, SBC and STA, two different modes of indirect addressing are provided: preindexed indirect, and post-indexed indirect. Pure indirect addressing can be obtained, using either mode, by first setting the respective index register to zero.

#### *Pre-indexed indirect addressing*

Examples:   STA (zerotable,X)  
              EOR (&60,X)

This mode of addressing is used when a table of effective addresses is provided in zero page; the X index register is used as a pointer to select one of these addresses from the table.

In pre-indexed indirect addressing the second byte of the instruction is added to the X register to give an address in zero page. The two bytes at this zero-page address are then used as the effective address for the instruction.

#### *Post-indexed indirect addressing*

This indexed addressing mode is like the absolute,X or absolute,Y indexed addressing modes, except that in this case the base address of the table is provided in zero page, rather than in the bytes following the instruction.

In post-indexed indirect addressing the second byte of the instruction specifies a zero-page address. The two bytes at this address are added to the Y index register to give a 16-bit address which is then used as the effective address for the instruction.

Examples:   ADC (&66),Y  
              CMP (pointer),Y

This last addressing mode is very useful. An example of its use is given in the program below, which will only work on a machine without a second processor attached. It clears the screen.

```

10 DIM MC% 100
20 addr=&70
30 FOR pass =0 TO 2 STEP 2
40 P%=MC%
50[OPT pass
60.cls
70  LDA #&58  High byte of start address
80  STA addr+1
90  LDY #0    Low byte of address,
100 STY addr  and value to write to screen
110 TYA      Put zero (black) into A
120.clsloop
130 STA (addr),Y store zero
140 INY
150 BNE clsloop do 256 times
160 INC addr+1 Increment hi byte of address
170 LDX addr+1 Set status register to addr
180 BPL clsloop Compare with top of RAM
190  RTS
200]
210 NEXT pass
220 MODE 4
230 COLOUR 129
240 CLS      White out screen
250 A=GET    Wait for a key
260 CALL cls Black out screen
270 COLOUR 128
280 END

```

## 6. THE STACK

The 6502 processor supports a hardware stack. This is an important part of the computer which can be used both by the CPU and the programmer. This chapter looks at how the stack can be used and how it performs its task.

### 6.1 Using the stack

A hardware stack is simply a set of memory locations (&100 to &1FF) which are reserved by the processor. These locations can be used as temporary storage locations. Up until now, when we have wanted to store a value, we might have used.

STA tempaddr

And to recall the value which was in the accumulator at that time, the instruction

LDA tempaddr

Loading the accumulator with the value from 'tempaddr' does not alter the value stored in 'tempaddr'. Hence the number may be recalled several times. When storing a value on the stack, however, the situation is different. The value to be stored is 'pushed' onto the stack and when it is wanted again, it is 'pulled' off into a register - a one-time only operation.

The stack is a LIFO structure, these initials stand for 'last-in, first-out', which means that the first item put on the stack will be at the bottom and so will be the least readily available, whereas the last item on the stack will be at the top and will be the first one to be pulled off it.

The four instructions which a programmer can use to access the stack are:

*Mnemonic*    *Description*

PHA	push the contents of A onto the stack
PLA	pull a value off the stack and store it in A
PHP	push the contents of the status register P
PLP	pull a value off the stack and store it in P

'Last-in, first-out': the stack forbids the use of any item other than in that order.

A stack pointer is used to manage the stack. This is a register which contains the address of the top location being used. For example, on encountering a PLA instruction, the accumulator is loaded from the memory location pointed to by the stack pointer and then the stack pointer is automatically moved back one location. Whenever a PHA instruction is executed, the accumulator is stored in the memory location pointed to by the stack pointer and then the stack pointer is moved on to the next location.



*Example*

The following series of pushes and pulls leaves the stack in a state which is shown below:

```
LDA #78
PHA
LDA #79
PHA
PLA
LDA #80
PHA
```

	Stack pointer points to next free space
80	Last value pushed onto the stack
78	Previous value pushed onto the stack

Note that the value 79 which was pushed onto the stack and then pulled off it again no longer occupies a location on the stack.

To see why the stack is used as a temporary storage place in preference to a memory location such as 'tempaddr', consider the two alternative sections of assembler below:

```
PHA          STA tempaddr
LDA addr    LDA addr
CLC         CLC
ADC #12     ADC #12
STA addr    STA addr
PLA        LDA tempaddr
```

These both produce the same result when executed but the one on the left will produce less code. This is because the LDA and STA instructions consist of either two or three bytes, one for the op-code and one or two for the address. The PHA and PLA instructions, however, just consist of an op-code.

Note that if more than one value is stored on the stack at once, care must be taken when these values are retrieved. Because it is a LIFO structure, the values must be taken off the stack in the opposite order to how they were placed on it, e.g.

```
PHA  save accumulator
TXA  prepare to save X
PHA  save X
PLA  restore value
TAX  transfer to X the last value pushed
PLA  restore accumulator
```

When using the stack it is very important to pull as many values as you push. Otherwise confusion can arise as we will see below.

## 6.2 How the CPU stores addresses

The stack is used by the CPU as well as by the programmer. On encountering a JSR instruction, the address of the instruction following the JSR is stored so that the CPU knows where to start executing from when it comes to the end of the subroutine. This is done by pushing the two bytes of the address onto the stack so that they can be retrieved when the RTS is reached. Thus a routine which is entered with a JSR and finishes with RTS should always pull the same number of bytes as are pushed otherwise the value obtained from the top of the stack by the RTS will not be the correct return address, e.g.

```
.entersubroutine
PHA
LDA addr
CLC
ADC #12
RTS
```

When the RTS is reached the CPU will pull two values off the stack, and put them into the program counter. However, as one of these values is the value pushed with the 'PHA', the program counter will almost certainly contain the wrong address. This will mean that the CPU will start trying to execute instructions at the wrong address and do something undefined by the designers of the 6502.

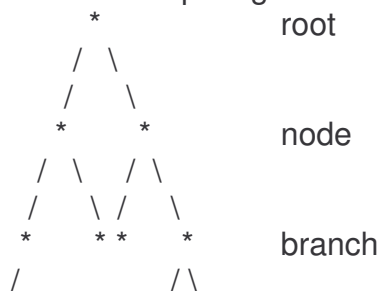
## 6.3 Recursion

One of the most important reasons for using a stack to hold the addresses of subroutine returns is that recursion is then automatically supported.

A recursive subroutine is one which calls itself. This can be a very powerful feature and enables a programmer to implement tree structures as shown below.

Using trees

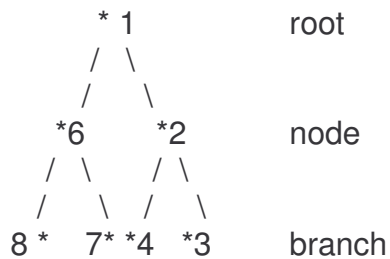
A tree in computing is normally pictured as follows:



Note that it is usually drawn with the 'root' at the top.

In order to print out all the elements (root and nodes) in the tree, you must write a routine which prints out an element and then goes down a branch to the element beneath it. If there isn't an element below, then it goes back up one level and sees if there is an alternative branch from there. For the above tree the order that the elements would be visited, assuming that the routine shows a preference for right-

hand branches, is:



A BASIC routine to do this is as follows:

```
DEFPROCtree (element)
PRINT value(element)
IF right(element) THEN PROCtree(right (element))
IF left(element) THEN PROCtree(left (element))
ENDPROC
```

This assumes that each element has three things known about it: its value, the element that its right branch leads to (FALSE if no branch) and the element that its left branch leads to (FALSE if no branch). These should be stored in three arrays whose names are 'value', 'right' and 'left'.

The routine can be called using

```
PROCtree(0)
```

The assembler version of this is:

```
.enter      LDX #0           Start at root (zeroth element)
.tree      LDA value,X      Get value of element
           JSR printnumber  See section 13.1 for details
           LDA right,X      Is there a right branch?
           BEQ tryleft     If not, try a left one
           TAX
           JSR tree        Else take that branch
.tryleft   LDA Left,X       Is there a left branch?
           BEQ backup     If not, go back up one level
           TAX
           JSR tree        Else take that branch
           RTS            Return
```

In this case the block of memory locations starting with the address 'value' should contain the values of each element in turn: those starting at 'right' should contain the element to which each one's right-hand branch leads, and 'left' the element to which each one's left-hand branch leads.

## 7. MACROS

If an assembler programmer wants to use a block of instructions several times during a single program then this block only needs to be entered once. There are then two methods of using this block when it is needed. The first is to put a label before it and an RTS instruction at the end, and reference it as a subroutine using the JSR instruction which was described earlier. In this case the CPU will 'jump' to the label when it reaches the JSR and 'jump' back again when it reaches the RTS at the end of the subroutine.

The alternative method is to turn the block of instructions into a macro, the method for doing this will be explained later. Essentially what this does is to give this block of instructions a name and, when the assembler comes across this name, it inserts the instructions of the macro into the object code which it is producing so that the CPU does not have to perform any jumps when it is executing the code.

Hence the main difference between macros and subroutines is that subroutines are called at run-time and macros are called at assembly time.

The advantage of macros is that they are faster than subroutines since no jumps are needed during execution. The disadvantage is that if the macro is to be used several times, the resulting machine code program will have to contain multiple copies of the instructions represented by the macro, and thus be long. Using a subroutine would only require one copy of the instructions. Macros can be more useful than just an aid to save typing however, and this chapter explains some of their other features. Further examples can be found in section 12.5 (General purpose macros).

### 7.1 Generating and calling a macro

Consider the sequence of instructions:

```
ROR A : ROR A : ROR A : ROR A
```

This simply shifts the upper nibble of the accumulator into the lower nibble. A macro with the name 'FNrotateacc' containing this sequence can be set up outside the assembler program as follows:

```
DEF Fnrotateacc  
[OPT pass  
ROR A : ROR A  ROR A : ROR A  
] :=pass
```

This macro can then be called from inside the assembler with the statement  
OPT FNrotateacc

The OPT statement is being used here as a dummy statement, simply to call FNrotateacc and have no other effect. We therefore arrange for the value of the

function to be 'pass', which should be the value used in the initial OPT statement. Therefore on reaching this statement the assembler will generate the machine code corresponding to the assembler instructions of the macro, and place this in the object code. Then it will move on to the next instruction of the assembler program.

The flow of control when the program is being typed will look something like this:

```
LDA addr
FNrotateacc
STA addr
```

The flow of control when the program is being assembled will look something like this:

```
LDA addr
DEFFNrotateacc
ROR A
ROR A
ROR A
ROR A
:=pass
STA addr
```

The machine code produced will be as follows:

```

A5  81  LDA addr  addr  &81
6A           ROR A
6A           ROR A
6A           ROR A
6A           ROR A
85  81  STA addr
```

## 7.2 Macro parameters

Macros can take parameters, thus the previous example could be rewritten in such a way that it could rotate the accumulator any number of times (as long as this number is greater than 0):

```
DEF FNrotateacc(rotate)
FOR number 1 TO rotate
[OPT pass
ROR A
]
NEXT number
= pass
```

So, to rotate the bits in any memory location any number of times to the right, simply set up a macro as follows:

```
DEF FNrotate(address, rotate)
FOR number = 1 TO rotate
```

```
[OPT pass
ROR address
]
NEXT number
= pass
```

A typical call might be

```
OPT FNrotate(&3000, 4)
```

This would generate machine code to rotate right four times the bits in location &3000.

### 7.3 Conditional assembly in macros

Macros can also be constructed to contain conditional instructions, so that they will assemble different pieces of code according to the parameters passed. For example, the following macro works out the shortest way of rotating the accumulator left:

```
DEF FNoptimumrotate(rotate)
IF rotate < 1 THEN = pass
IF rotate < 5 THEN FOR number = 1 TO rotate
[OPT pass : ROL A:]:Next number
ELSE FOR number = 1 TO (9-rotate):
[OPT pass : ROR A:]:Next number
= pass
```

### 7.4 Labels in macros

Labels cannot be used in the normal way inside macros. Consider, for example, the macro given below:

```
DEFFNstar
[OPT pass
LDX addr
BEQ exclamation
LDA #ASC"*"
JSR oswrch
.exclamation
LDA #ASC"!"
JSR oswrch
] : =pass
```

When the assembler reaches the 'BEQ exclamation' loop instruction on the second pass, it will give the offset the same address of the label which was set up the first time around. If forward referencing is not used then this problem will not occur, but it is still undesirable to use label names time and again. The program becomes very difficult to follow and to debug.

To use labels in macros that are called from more than one place, it is necessary to set up tables of labels. Thus if the label 'start' is used in a macro, an array called 'start' would have to be DIMensioned at the beginning of the program together with as many elements as the number of times the macro is called. If the macro with 'start' in it was called three times from within the program, the statement 'DIM start(2)' would have to be inserted before the first call to that macro took place. Also, each call to the macro would have to pass a parameter which contained a number (0,1 or 2) so that the correct label would be used. To illustrate:

```
DEF FNmacro(fred,jim,no)
[OPT pass
LDA fred
LDY jim
.start (no)
JSR oswrch
DEY
BNE start(no)
]
=pass
```

The above macro will print the contents of 'fred' as an ASCII character, 'jim' times. By convention the label number is always passed as the last parameter, and it is also a good idea to have all the macros using the same variable to hold the number (in this case 'no').

A typical call to the above macro might be

```
OPT FNmacro(addr, 45, 1)
```

This would use the label 'start(1)'.

## 8. BASIC I, BASIC II AND ELECTRON BASIC

There are, at the time of writing, two versions of BBC BASIC available for the BBC Microcomputer, known officially as BASIC and BASIC II. In this book, however, they are referred to as BASIC I and BASIC II respectively, and the name BASIC is used as a general term to cover either. This chapter looks at the differences between the two which affect the assembler programmer and provides BASIC I versions of the new directives and keyword.

The BASIC provided on the Electron can be assumed to be BASIC II. The BASIC routines given elsewhere in this book will all work on machines containing BASIC II and can be adapted to work with BASIC I as well.

### 8.1 Distinguishing BASIC I from BASIC II

To find out if you have BASIC I or BASIC II press BREAK and then type REPORT. BASIC II will give the message:

(C)1982 Acorn

Whereas BASIC I will produce:

(C)1981 Acorn

### 8.2 The main differences between BASIC I and BASIC II

The main differences between the two BASICs which concern assembler programmers are:

#### OPT

In BASIC I only the lowest two bits in the OPT statement are significant; if the lowest bit is set then the machine code is listed and if the next bit is set error messages are reported. The other bits are ignored. However, in BASIC II the third bit is significant as well; it is used to produce code which will execute somewhere other than the assembled position. If the third bit is set (OPTs 4 to 7) then the code is assembled at the value of O% (the code origin), not at P%. However, all the JMP's etc. will be set up as if it is going to execute at P%, and so it is an easy matter to relocate the code. This is particularly useful if, for example, you had written a routine which had to work in ROM, or some other space which is not normally accessible. Note that if this option is used then as the code is produced both O% and P% are incremented, otherwise just P% is incremented.

#### EQUB, EQUW, EQUD and EQU S

Four assembler directives have been introduced in BASIC II which are not present in BASIC I. These new directives are EQU B, EQU W, EQU D and EQU S which stand for 'equate byte', 'equate word' (2 bytes), 'equate double word' (4 bytes) and 'equate string' (0 - 255 bytes). These each take a single argument, and put its value into the assembly code at P% (also incrementing P% by the correct amount), e.g.



EQUB &FE	Put '&FE' at P%
EQUW oswrch	Put contents of 'oswrch' at P% and P%+1
EQUD 0	Set the next four bytes to zero
EQU\$ "Fred"+CHR\$(13)	Put 'Fred'+Carriage Return in memory starting at P%

### OSCLI

A new keyword, OSCLI, has been introduced which takes as its argument an expression, which it then passes to the operating system command line interpreter. This is not directly useful in assembler source code, but is useful when saving or loading variable amounts of data, or when sending variable FX commands. For example the following routine sets up soft key 0 to contain the string 'LIST+ERL+[RETURN]':

```

PROCkey (0, "LIST "+STR$ERL+"|M")
*
*
DEFPROCkey(number, A$)
OSCLI("KEY " + STR$number + " " + A$)
ENDPROC

```

Another example is to SAVE a BASIC program by typing

```
OSCLI("SAVE <filename> " STR$~PAGE + " " + STR$~TOP)
```

Note: this is exactly the same as the BASIC command SAVE <filename>. Note also the use of 'STR\$~' here to convert a hexadecimal number to a string.

## 8.3 BASIC I versions of EQUB, EQUW, EQU\$ and EQU\$

Macros can be set up in BASIC I to emulate these directives:

### EQUB

```

DEF FNequb(byte)
?P% = byte
P% = P% + 1
= pass

```

### EQUW

```

DEF FNequw(word)
?P% = word AND &FF
P%?1 = word DIV &100
P% = P% + 2
= pass

```

## EQUd

```
DEF FNequd(doubleword)
!P% = doubleword
P% = P% + 4
= pass
```

## EQUs

```
DEF FNequs(string$)
$P% = string$
P% = P% + LEN(string$)
= pass
```

Note that 'FNequs' will put the string into memory from 'P%' on, and will also set the byte following the string to a &D byte (RETURN character), although the next mnemonic assembled will overwrite this. In the event that this is a problem, 'FNequs' could be rewritten so that only the string is put into memory, and nothing more. This is left as an exercise for the reader.

## 8.4 BASIC I version of OSCLI

The following routine can be used in exactly the same way as OSCLI is used in BASIC II, e.g.

```
PROCoscli("SAVE <filename> " + STR$~PAGE + " " + STR$~TOP)
```

Note that 'cli' is an ATOM string which should be DIMensioned at the start of the program, e.g. DIM cli 64

```
DEFPROCoscli ($cli)
LOCAL X%, Y%
X% = cli AND &FF
Y% = (cli AND &FF00) DIV &100
CALL oscli                      oscli is at &FFF7
ENDPROC
```

## 9. OS ROUTINES AND SPECIAL EFFECTS

A whole book would be needed to describe all the features of the operating system and the routines it contains. This chapter briefly introduces two operating system calls, 'OSBYTE' and 'OSWORD'. Between them, they perform a wide variety of tasks. It then shows how operating system routines can be intercepted and replaced by user defined ones. Finally it takes a look at some of the special effects which can be obtained using either the operating system commands or specialised hardware provided by the BBC Microcomputer and Acorn Electron.

### 9.1 OSBYTE and OSWORD

#### OSBYTE - &FFF4

OSBYTE calls can be used to access several operating system routines. The particular routine is selected by the number passed in the accumulator. The X and Y registers are used to pass any parameters needed to the routine and to pass back any results which may be produced as a result of the call, e.g.

```
LDA #12  
LDX #10  
JSR osbyte
```

will call OSBYTE 12 which sets the keyboard auto repeat rate, in this case to 10 centiseconds.

```
LDA #129  
LDX #&9D  
LDY #&FF  
JSR osbyte
```

will call OSBYTE 129 which performs the INKEY function, in this case it is being called with a negative value (&9D = -99) and performs a keyboard scan to see if the key with this value, the Space bar, is being pressed.

On exit, X and Y contain &FF if the key being scanned was pressed and 0 otherwise.

\*FX calls are used to access OSBYTE calls from BASIC. In this case the values of the registers are passed in the following way:

```
*FX 12,10
```

This will have the same effect as the first example.

However \*FX calls do not return results so it is not appropriate to replace the second example given by a \*FX call.

## **OSWORD - &FFF1**

OSWORD routines are similar to OSBYTE routines, the difference being that parameters are not passed in the X and Y registers, instead they are passed in a parameter block, and the X and Y registers are used to contain the address of this block, e.g.

```
LDA #2
LDX #&80
LDY #&00
JSR osword
```

This calls OSWORD 2 which sets the value of the system clock to the five byte value which is stored in memory starting at the address &0080.

```
LDA #1
LDX #&80
LDY #&00
JSR osword
```

This calls OSWORD 1 which reads the system clock, the five byte value being returned in memory starting at the address &0080.

## **9.2 Revectoring operating system routines**

Many of the operating system routines are not entered directly by jumping to their position in the ROM, instead they are entered via addresses stored in the RAM.

For example, to use the operating system write character routine (OSWRCH) (the instruction which has been used in previous examples is JSR &FFEE). Location &FFEE, however, contains not the start of the routine, but the instruction JMP (&20E), since the address of the code for OSWRCH is stored in locations &20E and &20F in RAM. These locations are known as the 'vector' for this routine.

Accessing routines indirectly, via vectors in the RAM has several advantages. In different operating systems the entry position of the routine may alter, but this will not affect the user since the instructions JSR &FFEE or JMP (&20E) will still access it. The difference will be dealt with by the operating system which will store the correct addresses in locations &20E and &20F.

In addition the user can intercept any of the routines by 'revectoring' them. For example he could change the contents of &20E and &20F so that they contained the address of a user defined routine. One use of this is shown below.

### *Pretty Printer – prettyprint*

When printing text to the screen it is often difficult to ensure that words will not be broken at the end of the line. The following routine achieves this. When linked in, it will buffer characters up to a space or carriage return character, and then only output the characters on the same line if there is room without splitting them. Note that the routine does not deal with control characters (codes less than 32) that have

trailing characters. The routine should be linked into the OSWRCH vector at &20E - &20F, by typing !&20E = !&20E AND &FFFF0000 OR prettyprint

Notice that the variable 'linelength' is set to the length of the line (19, 39 or 79, depending on the screen mode selected). On entry A holds the character to be printed. X and Y are irrelevant. A typical call would be any call to OSWRCH. On exit all registers have been preserved.

An example of the output of this is: (40 column screen)  
 This text is Prettily Printed This text is Prettily Printed This text  
 Is Prettily Printed This text is Prettily Printed This text is  
 Prettily Printed this text is prettily Printed

```
.prettyprint
    PHA
    STX savedx      Save X register
    LDX pointer     Get line pointer
    CMP #ASC" "     Is it a space ?
    BEQ isspace
    STA buffer,X    Store character
    INX             Increment pointer
    CPX # linelength Is buffer full ?
    BNE exit        If not, get next character
    STX pointer     Update pointer
    BEQ newline     Branch always

.isspace
    CPX #0
    BEQ exit
    JSR getpos
    LDX #0          set X to zero for printbuffer
    LDA pos         get cursor position( x-coord)
    CLC
    ADC pointer     get cursor x+pointer
    CMP # linelength If >= linelength
    BCS newline    print buffer
    LDA pos         If cursor is at .beginning
    BEQ printbuffer of a line print the .buffer
    LDA #ASC " "   else print a space
    JSR printchar
    JMP printbuffer and print the buffer

.newline
    LDA #13
    JSR printchar
    LDA #10
    JSR printchar

.printbuffer
    LDA buffer,X   get characters
    JSR printchar  print characters
    INX            increment pointer
    CPX pointer    if line pointer<> line end
```

```

        BNE printbuffer    then get next character
        LDX #0
.exit
        STX pointer        save pointer
        LDX savedx         restore X register
        PLA                restore A
        RTS                Return
.getpos
        TYA                save Y
        PHA                push to stack
        LDA #&86           osbyte 86 is read cursor position
        JSR osbyte         return pos and vpos in X and Y
        STX pos            X holds x pos of cursor
        PLA                restore Y
        TAY
        RTS
.printchar
        JMP (oldoswrch)    old oswrch holds original contents of &20E and &20F

```

### 9.3 Screen Scrolling

On both the BBC Microcomputer and the Acorn Electron, there are two screen-scrolling methods known as software scrolling and hardware scrolling. Software scrolling is often slow. If you define a text window to cover the whole screen (VDU 28,0,24,39,0 in MODES 6 or 7 only) and then scroll and screen (by moving the cursor off the bottom of the screen), you will notice the scrolling slowing down as it attempts to move all the screen memory up a line. An alternative and faster method of scrolling has been incorporated in the hardware.

A section of each computer incorporates a register designed to hold the start of screen memory. In the BBC machine it is the 6845 CRTIC (Cathode Ray Tube Controller), and in the Electron it is a section of the ULA (Uncommitted Logic Array). To employ this screen-scrolling method, it is only necessary to change the number in the register. On the next vertical sync, the new screen will be displayed starting at that number. To scroll the screen up on the BBC machine, simply type:

```

MODE 6
VDU 23; 12, &0C; 0; 0; 0; : VDU 23; 13, &28; 0; 0; 0;

```

and on the Electron

```

MODE 6
?&FE02 = &A0 : ?&FE03 = &30

```

To explain: on the BBC machine there are in fact two registers which control hardware scroll. These are registers 12 and 13. Register 12 contains the high byte of the start address, and register 13 contains the low byte. Things are not quite this simple however, as the start address held in the two registers is only to the nearest 8 bytes (1 character cell in the MODEs 0 to 6), and so the number put into the registers is the start address, DIV 8. In the above example, the new address is

$\&6000 + 40 * 8 (= \&6140)$  DIV 8, which is  $\&C28$ , and so we put  $\&C$  in register 12 and  $\&28$  in register 13.

On the Electron things are not quite the same. The address held in the hardware is not to the nearest 8 bytes, but to the nearest 64 bytes.

The value to put into the Electron's ULA is the address of the top of the screen, divided by 2. The two registers are at  $\&FE02$  and  $\&FE03$  (low byte and high byte). The address,  $\&6140$ , is written there by working out  $\&6140$  DIV 2 ( $= \&30A0$ ), and then writing the low and high bytes of the new value into the registers.

You will have noticed that the hardware scroll operation is not exactly the same as that of the operating system scroll, in that the top line is filled not with spaces but with what was on the bottom line when the process began. This is because the memory map will 'wrap round', as in the following diagram;

	0	1	2	3		37	38	39
0	$\&6140$	$\&6148$	$\&6150$	.....	.....	$\&6268$	$\&6270$	$\&6278$
1	$\&6141$	$\&6149$	$\&6151$	.....	.....	$\&6269$	$\&6271$	$\&6279$
2	$\&6142$	$\&614A$	$\&6152$	.....	.....	$\&626A$	$\&6273$	$\&627B$
3	$\&6143$	$\&614B$	$\&6153$	.....	.....	$\&626B$	$\&6273$	$\&627C$
4	$\&6144$	$\&614C$	$\&6154$	.....	.....	$\&626C$	$\&6274$	$\&627C$
5	$\&6145$	$\&614D$	$\&6155$	.....	.....	$\&626D$	$\&6275$	$\&627D$
6	$\&6146$	$\&614E$	$\&6156$	.....	.....	$\&626E$	$\&6277$	$\&627E$
7	$\&6147$	$\&614F$	$\&6157$	.....	.....	$\&626F$	$\&6277$	$\&627F$
8	$\&6280$	$\&6288$	$\&6290$	.....	.....	$\&64E8$	$\&64F0$	$\&64F8$
	.....	.....	.....	.....	.....	.....	.....	.....
$24*8+6$	$\&7F46$	$\&7F4E$	$\&7F56$	....	....	$\&606E$	$\&6076$	$\&607E$
$24*8+7$	$\&7F47$	$\&7F4F$	$\&7F57$	....	....	$\&606F$	$\&6077$	$\&607F$

## 9.4 Palette handling

Both the BBC Microcomputer and the Acorn Electron provide a palette facility in the 'soft' screen modes. On the Electron all modes are 'soft' screen modes, but on the BBC machine Teletext (MODE 7) has no palette facility. The idea is that each mode can display a certain number of colours at any one time (16 in MODE 2, 4 in MODE 5 and so on).

Essentially the palette provides a mapping between the screen memory and what appears on the screen. The screen memory contains logical colours, and these are represented by the palette as physical colours which you see displayed on the screen. Thus in MODE 1, where there are four logical colours, each can be represented as any of the sixteen physical colours which these computers are capable of producing. Use the VDU19 statement to tell the computer how to represent a particular logical colour as a particular physical colour, i.e.

VDU 19, logical colour, physical colour ; 0 ;

For example

VDU 19,1,3;0;

This tells the computer to display all occurrences of logical colour 1 in its memory as physical colour 3 (Yellow). Think of the palette as a mapping of physical colours onto logical colours, where all that the VDU 19 statement does is to simply change the mapping. The illustration below should make this clear:

<i>Colour Mode</i>	<i>Logical Physical number</i>	<i>colour</i>
0,3,4,6	0	black
	1	white
1,5	0	black
	1	red
	2	Green
	3	Yellow
	4	Blue
	5	Magenta
	6	Cyan
	7	White
	8	F(lashing) black / white
	9	F red / cyan
	10	F green / magenta
	11	F yellow / blue
	12	F blue / yellow
	13	F magenta / green
	14	F cyan / red
15	F white / black	

Another way of changing the palette is to call OSWORD with A set to 12. In this you simply set up a block of 5 bytes to this format:

Paletteblock	logical colour
paletteblock+1	Physical colour
paletteblock+2	0
paletteblock+3	0
paletteblock+4	0

Then OSWORD is called in the normal manner (with X and Y pointing to the parameter block, 'paletteblock' in this case). This has precisely the same effect as VDU 19, except that it is faster and also may be called from an interrupt or Event routine (See Interrupts below).

OSWORD 12 is not available on BBC machines with OS 0.10.



## 9.5 Interrupts, events and BREAKs Interrupts

Both the BBC Microcomputer, and the Acorn Electron run under interrupts. Interrupts allow the machine to update its own internal variables, without the user even realising that their program is not in complete control.

On the 6502 there is an interrupt request pin (IRQ) which, when a signal hits it, tells the processor that an interrupt request has occurred. The 6502 then has the option of ignoring the interrupt. This decision is made by the state of an interrupt flag. If the flag is set, then the interrupt will be ignored, otherwise the operating system will deal with it.

The interrupt flag can be altered with the two assembler instructions:

<i>Mnemonic</i>	<i>Description</i>
SEI	set interrupt disable flag
CLI	clear interrupt disable flag (Default state)

Note that the interrupt flag should not really be altered, as then all interrupt driven devices (keyboard, flashing colours, sound, etc.) would stop working.

There are two interrupt vectors provided by the operating system. These are IRQ1V (at &204), through which all interrupt requests are passed, and IRQ2V (at &206), through which any unrecognised interrupts are passed. Normally, IRQ2V would be used, but if you want to update your own device before the operating system can act, then you should use IRQ1V. The routine must first handle the interrupt, then disable the device that caused the interrupt, and finally, it must perform a 'JMP (oldIRQ1V)' (where 'oldIRQ1V' is the old contents of IRQ1V). This should only be used if there is no other way of achieving the desired effect.

One way to set up interrupts on the BBC Microcomputer is by the User 6522 VIA (Versatile Interface Adapter). This has two timers, which can be set to count down from any particular 16-bit value, and to cause an interrupt request on reaching zero. Also, it will be necessary to write a routine to handle this, and to put the address of the entry point of the routine in the correct vector (in this case IRQ2V). Note that the routine must perform an 'RTI' in order to transfer control back to the operating system. RTI stands for return from interrupt.

All this might seem a bit messy, and so a second kind of interrupt peculiar to the BBC microcomputer and Electron has been implemented. This second kind of interrupt is called an Event. (It is not implemented on BBC Microcomputers with OS 0.1).

### Events

These operate in a similar way to interrupts in that they are totally transparent (undetectable by the user program) and are indirect, via a vector (at &220).

Certain occurrences within the machine have events associated with them, and these events can be trapped by the user. These are:

- 0 - Buffer empty, where X gives buffer identity
- 1 Buffer full, where X gives buffer identity and Y holds character that could not be stored.
- 2 Keyboard interrupt
- 3 ADC conversion complete
- 4 Start of TV field pulse (vertical sync)
- 5 Interval timer crossing zero
- 6 Escape condition detected
- 7 RS423 receive error
- 8 Remote procedure call detected (on Econet)

Events can be selectively disabled and enabled with OSBYTEs 13 and 14, where X specifies the event. Note that the default state is all events disabled.

Example event handler:

```

10 REM Event handler
20
30 vsynccounter=&70
40 DIM code 100
50 FOR pass = 0 TO 2 STEP 2
60 P%=code
70[ OPT pass
80.event
90 PHP preserve status
100 CMP #4 is this the event we want
110 BNE notvsync if not return
120 INC vsynccounter increment event counter
130.notvsync
140 PLP restore status
15] RTS return
16O]
170 NEXT pass
180
190 eventvec = &220
200 ?eventvec = FNlo(event)
210 eventvec?1= FNhi(event)
220 *FX 14 4
230 END
240
250 DEF FNlo(value) = value AND &FF
260
270 DEF FNhi(value) = (value AND&FF00)DIV 256

```

## BRKs

The 6502 supports a BRK instruction. This generates a software interrupt, which is similar to the interrupt request described earlier, except that it cannot be disabled. BASIC and the operating system use BRKs for flagging errors. This means that the

BRK handler will print an error message. The standard format of the BRK error message is;

BRK instruction (op-code is &00)  
Fault number (one byte)  
Fault message (string of characters terminated by a zero byte)

Thus it is possible to put error messages in a program, and have them printed out by the BRK handler (which, incidentally, is normally handled by the language). This can be useful for debugging purposes. A useful macro for this is:

```
DEF FNerror(err, error$)
[ OPT pass
BRK      Cause BREAK
EQUB err  Fault number
EQU$ error$ Error message
EQUB 0    Message terminator
.pass
```

A typical call to this would be:

```
OPT FNerror(60, "Hello")
```

where '60' is the fault number, and 'Hello' is the message to be printed when that BRK is activated.

It is, of course, possible to write your own BRK handler, by simply putting the start address of a suitable routine in the BRK vector (&202). For example, the following routine prints out all registers at a BRK:

```
0REM BREAK Handler
10
20 oswrch = &FFEE
30 osnewl = &FFE7
40 stringptr=&70
50 exit=stringptr
60 temp=stringptr+2
70 exit = !&202
80 DIM code 200
90 FOR pass = 0 TO 2 STEP 2
100 P% = code
110[ OPT pass
120.header
130 EQU$ " A X Y PC N V U B D I Z C"+ CHR$10 + CHR$ 13
140.break
150 TYA X and Y
160 PHA push all registers so they may be printed
170 TXA
180 PHA
190 LDA &FC Get accumulator
```

200	PHA	
210	JSR osnewl	go onto a new line
220	LDX #FNlo(header)	Print "A X Y PC..
230	LDY #FNhi(header)	
240	JSR atomstring	
250	PLA	Get accumulator
260	JSR hexandspace	print it and a space
270	PLA	Get X register
280	JSR hexandspace	Print it and a space
290	PLA	Get Y register
300	JSR hexandspace	print it and a space
310	LDA &FE	FE an FD hold the
320	JSR printhex	program counter where
330	LDA &FD	the break occurred
340	JSR hexandspace	
350	PLA	Status register
360	JSR printbinary	Print P in binary
370	JMP exit	return to old error
380		
390.	hexandspace	
400	JSR printhex	Print A in hexadecimal
410	LDA #ASC" "	load a space
420	JMP oswrch	print it
430		
450	PHA	Save accumulator
460	LSR A	get
470	LSR A	top
480	LSR A	nibble
490	LSR A	
500	JSR print	print top hex digit
510	PLA	get bottom nibble
520	AND #&0F	
530.	print	print bottom hex digit
540	CMP #&0A	
550	BCC notalpha	if not 0 – 9 get char
560	ADC #6	carry is set here (add 7)
570.	notalpha	
580	ADC #&30	convert to ASCII
590	JMP oswrch	print and return
600		
610.	printbinary	
620	LDX #8	eight bits per byte
630.	binaryloop	
640	ASL A	get a bit
650	STA temp	print either 0 or 1
660	LDA #ASC"0"	
670	BCC printzero	
680	LDA #ASC"1"	
690.	printzero	
700	JSR oswrch	print 0 or 1

```

710 LDA #ASC" "           followed by a space
720 JSR oswrch
730 LDA temp
740 DEX
750 BNE binaryloop       Get next bit
760 RTS                   return
770
780.atomstring
790 STX stringptr        Address of string
800 STY stringptr+1      is given in X and Y
810 LDY #&100            Y is pointer along string
820.atomstringloop
830 LDA (stringptr),Y    get next character
840 JSR oswrch           print it
850 INY                  increment pointer
860 CMP #13              is it a return
870 BNE atomstringloop   if not repeat loop
880 RTS                   return
890
900]
910 NEXT pass
920 !&202=!&202 AND &FFFF0000 OR break
930[OPT 2
940.test
950 LDA #&01
960 LDX #&23
970 LDY #&45
980 SED
990 CLC
1000 BRK
1010 EQU 75
1020 EQU "HELLO"
1030 EQU 0
1040]
1050 CALL test
1060
1070 DEF FNlo(value)=value AND &FF
1080
1090 DEF FNhi(value)=(value AND &FF00)DIV &100

```

>RUN

```

A X Y PC  NVUB DIZC
01 23 45 1BB6 0 0 1 1 1 0 0 0
HELLO at line 1050

```

## 10. LARGE ASSEMBLER PROGRAMS

When writing substantial assembler programs it soon becomes evident that even 32K of memory is insufficient to hold both the assembler source text and the object code produced. This difficulty is heightened still further if the graphics modes are used. The problem can be overcome by breaking the source text into smaller modules or files and this chapter looks at how to set these up and how to use them.

### 10.1 Source files and the 'master compiler' program

A source module is a program which acts like a subroutine but has assembly code inside it. The master 'compiler' program reads in each source text module and assembles it. This program is shown below. (Note that anything enclosed within < > (angled brackets) is not to be typed in, but is to be replaced with the value for your application.)

```
0    REM Compile program
10   origin = <start of area for machine-code>
20   file$ = "ABC"
30   PROCrun("I",<page for source files>)
40   PROCrun("M",<page for macro file> (optional))
50   FOR pass = 0 TO 2 STEP 2
60   P% = origin
70   FOR files = 1 TO LENfile$
80   PROCrun(MID$(file$,files,1),<page for source files>)
90   NEXT files
100  NEXT pass
110  PRINT "Object code from &";~origin " to &";~P%
120  END
130
140  DEFPROCrun(name$,start)
150  PRINT name$;
160  OSCLI "LOAD SOURCE"+name$+" "+STR$~start
170  PAGE = start
180  GOSUB 0
190  ENDPROC
```

The files are assumed to be called SOURCEA, SOURCEB, ... i.e. the word 'SOURCE' followed by a single letter. The string 'file\$' holds the letters which identify them and hence in this example it contains 'ABC'. Since single letters are used, the length of 'file\$' gives the number of source programs. Note that in this example 'I' and 'M' should not be used for naming source files since they have their own special uses.

The program starts by reading in and assembling SOURCEI which is the initialisation file to be described later in this chapter. Then the macro file (if one exists) will be treated similarly; again this will be explained later. The main loop takes each of the source files in turn, loads it into the area you have defined as reserved for the source files, and then assembles it. For the first source file the

object code starts at the value of the variable 'origin', and P% is used by the assembler as a pointer to the next free byte. Hence this allows subsequent source files to be assembled so that their code follows on directly after that produced by the previous one.

A typical source file would be of the following format:

```
0    REM SOURCEX
10
20~OPT pass
30
...(Assembler text)
...
...
...
900
910] : RETURN
```

A typical memory map might look like this:

Start of BASIC	Screen - Smallest MODE possible (MODE 7)
	Variables shared by all source files
	COMPILE program
	Macro source file
	Source files
Origin	Object Code

Note that if you are using a 6502 second processor then the screen mode selected will not make any difference.

This method has been designed for use with disc based systems, but can also be used on cassettes if the tape is rewound between the two passes. To remind you of this you should place a 'Rewind Tape' message, together with a 'dummy=GET' statement (to wait for a key to be pressed as an indication that the tape is in the correct position), between the two NEXT statements.

## 10.2 Saving source files

One routine which is useful when using discs and the above method is a PROCsave routine;

```
DEF PROCsave OSCLI("SAVE <filename>+ STR$~PAGE + " "+STR$~TOP)
ENDPROC
```

Note: this routine will not work on BASIC I. See chapter 9 (BASIC 1, BASIC II and Electron BASIC) for a description of OSCLI and the equivalent BASIC I routine.

The routine should be inserted at the ends of all the source files, and called whenever you wish to save the source file that is in memory at the time.

Thus to SAVE the program all that is needed is to type 'PROCsave'. The reason this is so useful is that when editing large numbers of source files which all look alike, it is very easy to overwrite an existing file by typing the wrong name.

An alternative to this, which works on all versions of BASIC, is to type, in immediate mode, 'SAVE \$(PAGE+6)'. This looks at the first line of the program to find the filename. So, each program should start with

```
0    REM <filename>
100  <program>
200  ....
```

### Default soft keys

Another useful idea is to employ the machine's soft keys. This is best done by having a default soft keys program, which can be loaded at the beginning of the session.

```
10    REM Default Soft Keys
20
30    *KEY 0 I LLIST I N I M
40    *KEY 1 RUN | M
50    *KEY 2 LOAD"SOURCE
60    *KEY 3 CALLenter | M
70    *KEY 5 PROCsave I M
80    *KEY 6 PROCfind("
90    *KEY 7 MODE7 I M PAGE=&600 I L0AD "COMPILE" I M
100   *KEY 9 I L*CAT I M
```

Note that line 80 has a reference to PROCfind. This procedure is to be used from immediate mode to find all occurrences of strings in the current source module. (PROCfind is defined in section 12.6.) This procedure should be at the end of every source file.

### 10.3 Macro source files

If source files are to be used then calling macros can be a problem. To understand why, some knowledge of how BASIC works is required. When BASIC first comes across a reference to a function or procedure its search for the function or procedure definition starts from PAGE. Once it has found the definition it stores the address of the start of the function or procedure in memory, so that the next time it finds a reference to that particular function or procedure it doesn't have to waste time searching through the whole program again.



In the simplest case, when just one source file references a given macro, the macro can be added to the end of that source file, and the file treated normally. Consider, however, what would happen if two source files both had a reference to a macro called 'FNfred', and this macro was put at the end of each of them. Since they would almost certainly be different sizes the definition of 'FNfred' would, in each case, start at different addresses. Moreover, when the first source file was assembled the address of the macro in this file would be stored for use by all later references to the macro. Thus, when the second source file tried to use the macro, no searching through would occur. Instead the assembler would jump straight to the address which was stored by the first file and be unlikely to find 'DEFFNfred' starting there.

To avoid this problem a macro file is set up containing all the macros referenced in any of the assembler source files. This is present in the memory all the time, though in a different area of memory to the other files. Each of the macros must be called before the source files are assembled, however, so that the addresses where their definitions may be found are available to the compiler. Otherwise the first time that the compiler comes across a reference to a macro, e.g. FNfred, it will start searching for 'DEFFNfred', starting at PAGE, look through to the end of the source file, not find the definition, conclude that the macro doesn't exist and report 'No such FN/PROC'. So, set PAGE to the bottom of the macro file and call each macro once.

Thus a typical macro source file looks like this:

```
0    REM Macro file
10
20    pass = -1 : REM Dummy compilation
30    A% = Fmacro1(0,0) + Fmacro2(0,0,0,0)..
40    RETURN
50
60    DEF Fmacro(temp,no)
70    IF pass <0 THEN =TRUE
80[OPT pass
90 ...
```

Line 20 sets 'pass' to a value that the assembler will not use, so that the first time the macro is referenced it will not generate any code. Note that every macro in the file must have a test to see if it is being referenced for the first time (as at line 70).

Since the macro file needs to be resident in memory during the compilation a space will have to be assigned for it. This is usually between the top of the normal source files and the bottom of the COMPILE program.

## 10.4 Initialisation file

The initial file mentioned in COMPILE (line 30) is the file that sets up all variables to be used in the later source files. This is normally in the form:

```
0    REM SOURCEI
10
20    REM variables
30
```

-----  
This is where all the variables that will be accessed by all the source files are defined. Note that all the variables are defined to be resident in memory one after the other. Thus to move the block of variables around in memory, all you have to do is to simply change the value of 'P%' in line 40.

```
-----
40 P% = <start of memory to put variables>
50[OPT2 \ Report any errors, but don't list
60.<first variable> EQUB 0 \ Reserves one byte
70.<second variable> EQUW 0 \Reserves two bytes
30.<third variable> EQU$ STRING$(20,CHR$(0))
90 etc.
```

```
200]
```

-----  
This section sets up any constants that may be used in the program. The use of constants in any program is very important as is explained in chapter 12 (Program structure).

```
-----
210
220    REM Constants
230
240    limit = 45
250    numberofshapes = 12
260    oswrch = &FFEE
270    etc.
```

-----  
This section reserves memory for the data tables.

```
-----
300
310 REM Main Memory Allocations
320
330 P%=<start of memory allocated for tables>
340[OPT2
350.jim
360 OPT FNspace(80)
370.fred
380 OPT FNspace(300)          See section 12.5
390.length                  for FNspace
400 OPT FNspace(100)
410 etc
```

....  
....

---

This section fills the data tables.

---

```
600
610  REM fill data tables
620
630  FOR offset = 0 TO numberofshapes
640  READ offset?length, offset?width, offset?type
650  NEXT offset
660
670  DATA 20,12,3 ,36,24,1 ,etc.
680  etc
999  RETURN
```

# 11. PROGRAM STRUCTURE

The aim of this chapter is to help you write a large assembler program. It gives several tips how to produce structured and readable code which can be debugged easily.

## 11.1 Where to start

There are two entirely different approaches which can be used when producing a structured program. The first is to work out what the program is going to do and what it is going to look like before you start writing any code. Then you can start by writing the main loop which, for a game, could look something like this:

```
. enter
    JSR initialise
. restart
    JSR setupscreen
.main
    JSR plotshapes
    JSR checkcollisions
    JSR keyboardscan
    JSR updatecoordinates
    JSR checkifdead
    BNE main loop
    DEC lives
    BNE restart
    RTS
```

Although, at this stage, the subroutines have not been defined, it is obvious from their names what they are meant to do. The next task is to write these subroutines, again breaking them down into simpler routines if this is possible.

Since the main loop is the top level in the overall structure, this method is known as 'top-down' approach. Its advantage is that you know what you are aiming at right from the start. You shouldn't need any 'fixes' in the subroutines to make up for the fact that when you wrote them you didn't make them apply to all cases eventually required.

The alternative method is to start by writing the individual routines and then to add the code which joins them together into the final program. The advantage of this method is that all routines can be tested individually before very much effort has gone into the program. Consider how much time would be wasted if you used the first method and completed the whole program except for one routine which proved impossible to write in such a way that it performed satisfactorily.

In practice most people use a mixture of the two methods, so that any demanding routines are written first; then the top-level is written, followed by the rest of the program.

## 11.2 Self-documenting code

Long, directly referential variable names are a good idea when writing a program as they make the program more intelligible, e.g. 'JSR updatecoordinates' is better than 'JSR label2'. These may make the source code longer, but the problem can be overcome by splitting up the code into separate sections and compiling these individually as described in chapter 10 (Large assembler programs). Note that the fewer JMP's there are, the easier it is to follow the flow of control. Hence, using macros rather than subroutines also helps to increase the clarity of the code.

## 11.3 Parameters

Parameter passing is also recommended in assembler, for precisely the same reasons as it is in BASIC, viz. It enables a single block of code to be used for more than one purpose. However, the method of achieving this is somewhat different. Parameters are usually passed in the three registers A, X and Y, or alternatively the X and Y registers are used to specify an address of an information block, and the A register then holds some other information. A third method is to pass parameters in specified locations, so enabling an arbitrary number of parameters to be passed. This method doesn't support nesting or recursion, however.

### The CALL statement

This statement, which is used to transfer control from a BASIC program to a machine code program, can also pass parameters. When it is used, the bottom bytes of the BASIC variables A%, X% and Y% are transferred to the A, X and Y registers respectively. Also the lowest bit of C% is transferred to the carry flag.

Control is passed to the address given after the CALL statement and any parameters following this address are put into the parameter block starting at &600. The parameter block is of the following format:

- &600 Number of parameters
- &601 Parameter address (low byte)
- &602 Parameter address (high byte)
- &603 Parameter type
- &604 Parameter address (low byte)

The parameter types are as follows:

- Type 0 8-bit byte
- Type 4 32-bit word
- Type 5 40-bit floating point string
- Type 128 ATOM string
- Type 129 Microsoft string

In the case of a string parameter the address given points to a 'string information block', which contains the following:

Start address of the string

Number of bytes allocated  
Current length of string

An example of a CALL statement is

```
CALL enter,fred,A$
```

This would cause the machine code from 'enter' to be executed and the parameters 'fred' and 'AS' would be described in the parameter block as follows:

```
I 02 | C7 | 0E | 05 | DO | 0E | 81 | ?? | ?? | ?? |  
0600 0601 0602 0603 0604 0605 0606 0607 0608 0609
```

## The USR function

Another way of transferring control to a machine code routine is via the USR function. The differences between CALL and USR are that USR returns a result, a four-byte number consisting of the Status, Y, X and A registers (most significant byte to least significant byte), and takes only one parameter which is the address to which control is transferred.

### 11.4 Size of routines

All routines, whether in BASIC or assembler, should be as small as possible, so that they can be seen easily in their entirety. This is another real help when debugging. Debugging is often overlooked when estimating the time needed to write a program, and yet it is probably true to say that at least 50% of the time taken to write a program is taken up with debugging.

### 11.5 Conditional assembly as an aid to debugging

Conditional assembly can be used to insert extra instructions which print out intermediate values during debugging: these statements can be removed when the program is finally assembled. To do this a logical variable ('flag' in the following example) is given the value FALSE during debugging and TRUE otherwise. In the following example, if 'flag' is FALSE a routine to print the value of the accumulator in hexadecimal notation is assembled, and calls to this routine is inserted at two relevant points in the test program.

```
10 REM print hex digits  
20 DIM code 100  
30 oswrch = &FFEE  
40 FOR pass = 0 TO 3 STEP 3  
50 P% = code  
60[OPT pass  
70.enter CLC :ADC #&40 :]  
80 IF flag = FALSE [OPT pass : JSR debug:]  
90[OPT pass  
100 BEQ exit :SBC :#&10:]  
110 IF flag = FALSE [OPT pass : JSR debug:]
```

```

120 [OPT
130.exit RTS:]
140 IF flag THEN 360
150[OPT pass
160 # print hex digits
170.print
180 AND #&0F          get bottom four bits
190 CMP #&0A         if less than 10 then miss
200 BCC P% + 4       the next instruction
210 ADC #&06         Add 7 (6 + carry)
220 ADC #ASC"0"     Add ADC (0)
230 JMP oswrch      Write the character
240 # print A in hex
250.debug
260 PHA
270 PHA
280 LSR A           exchange top four bits
290 LSR A           for bottom four bits
300 LSR A
310 LSR A
320 JSR print      print first hex char
330 PLA
340 JSR print      print second hex char
350 PLA           restore original value of A
360 RTS           return
370]
380NEXT pass

```

The program works by finding out whether or not the four bits corresponding to each hex digit represent a number less than ten. If it is less than ten then the value ASC ("0") is added and the character, which will be a number 0... 9 will be printed. If it is greater than or equal to ten then a number equivalent to ASC ("A-10") is added so that ten will be printed as A, eleven as B etc.

For debugging purposes this program is assembled by typing

```

15 flag = FALSE
RUN

```

The program can then be executed for various values of A% by typing

```

A% = &12 : CALL enter

```

The final version of the program is assembled, without the debugging aids, by typing

```

15 flag = TRUE
RUN

```

## 11.6 Lower case variable names

As a convention, lower case is used for variable names. Most people consider this to make the code more readable. It also means that there is no chance of using variable names that conflict with BASIC's keywords e.g. 'print' can be used as a variable name, even though PRINT cannot).

## 11.7 Constants

Constants are used to give names to numbers which will be used several times throughout a program. They help to make the code easier to understand, e.g.

```
LDX #initial-lives
```

explains far better what is happening than

```
LDX #3
```

Using constants has another advantage - if a value needs to be changed and constants have been used, only the definition of the constant would need to be altered, rather than every occurrence of that value.

Some languages support constants and variables as totally different data types, and make it impossible to change the value of the constant. BASIC does not treat variables and constants differently (except for it's own constants, e.g. PI) and so it is up to the programmer to make sure that any constants defined retain their value. One convention used for this is to prefix all constants with a pound (£) sign as a reminder.



## 11.8 Lookup tables

Lookup tables are useful when converting one value to another. As an example consider the following:

```
LDY index  
LDA table, Y
```

In this example the value of A is dependent upon the value of Y. The example consists of a table of values, starting at 'table'.

Note that the above assembler is directly equivalent to the BASIC 'A=table?index'.

The values in 'table' could be the values of a palette, for example

```
.table  
    EQUB &3    #    Colour0    is    yellow  
    EQUB &1    #    Colour1    is    red (default)  
    EQUB &6    #    Colour2    is    cyan  
etc.
```

Thus lookup tables should be used wherever it is necessary to produce a value from another value, but only when there is no simple relationship between the two.

## 11.9 Use of absolute addresses

A mark of a good assembler program is that it will contain no absolute addresses in the assembler source code. Thus, if a data table starts at location &30F6, a constant should be set up initially to have the value &30F6, and the constant used in the assembler code, not the number &30F6. This follows on from the above points, and also makes the code easier to read and alter.

## 12. UTILITIES FOR ASSEMBLER PROGRAMS

This chapter consists of a number of routines which are designed to be used in any assembler program. Typical calls are given for each, as are the values to be passed to the routines in the registers, and those values to be returned by them.

Note that these routines are not meant to be complete programs and cannot be run without additional code being added, e.g. assignment of values to any addresses being used and all the necessary assembler directives. If you wish to use any of these routines inside your own programs then the comments to the right of the assembler statements may be omitted. If the routines contain any BASIC commands then any comments to the right of these must be left out or preceded by REM statements.

### 12.1 Input/Output

*Print BCD number – printnumber*

The 6502 microprocessor can perform arithmetic in two ways. These are binary (normal addition), and binary coded decimal (BCD). In the second method, each byte is split up into two nibbles, each of which can hold a decimal digit (0 to 9). Thus the largest number that can be held in one byte using this method is 99. The advantage of this method is that it is easier to output a binary coded decimal number in decimal than it is to output a straight binary number in decimal.

The following routine takes a BCD number in the accumulator and prints it, with leading zero suppression, at the current cursor position.

On entry, A contains the number to be printed, Y contains the leading zero flag (0 for no suppression, else suppress zeros), and X contains the ASCII code of the character to be printed in place of leading zeros.

A typical call to print out a two-byte BCD number, with leading zeros being replaced by spaces, would be:

LDX	#ASC"	Set Leading zero character to space
LDY	#&FF	Set Leading zero flag
LDA	highbyte	Get top byte of number
JSR	printnumber	Print it
LDA	lowbyte	Get bottom byte of number
JSR	printnumber	Print it

On exit X has been preserved and A and Y will have been corrupted.

.print number		
PHA		Save number
OPT	FNrotateacc (4)	Get top digit (See section 8.2)
JSR	printit	Print it
PLA		Restore number

	AND	#&0F	Get bottom digit
.printit			Fall through
	BNE	validchar	if non-zero print
	TYA		Check zero flag
	BNE	leadingzero	if set must be leading zero
.validchar			digit ok
	LDY	#&00	clear zero flag
	ORA	#ASC"0"	add in ASCII zero
	JMP	oswrch	print and return
.leadingzero			
	TXA		Get leading zero character
	JMP	oswrch	print and return

### *Keyboard scan – inkey*

This routine can be called to detect if a key is being held down at a particular instant; it uses INKEY of negative numbers.

On entry, X specifies the key to be tested. For details of the values for each key, see the table of INKEY negative numbers in Appendix A.

A typical call would be:

```
LDX #firstkey
JSR inkey
BNE keypressed
```

On exit, the zero flag is set or cleared depending upon the key's position at the time of testing. This routine does NOT go via the keyboard buffer. A and Y will have been preserved.

```
.inkey
    PHA                Save A
    TYA                Save Y
    PHA
    LDY #&FF           Negative numbers
    LDA #&81           osbyte &81 is INKEY
    JSR osbyte         Do it
    PLA                Restore Y
    TAY
    PLA                Restore A
    CPX                Adjust zero f lag
RTS                    and return
```

### *Sound*

Sound routines are always useful in games programs. The routine below works by using a table to hold all the sounds that it is to play, and, on entry, the number of the

sound to be played is given. To set up the table, the following could be used:

```
FOR offset = 0 TO <number of sounds> * 8 STEP 2
READ sndbuff!offset
NEXT offset
DATA 1,-15,200,20
DATA 3,1,150,10
DATA etc.
```

Where 'sndbuff' is the table to be filled with sounds, on entry, A holds the number of the sound to be played. X and Y are irrelevant.

A typical call would be:

```
LDA #firingsound
JSR sound
```

On exit, all registers will have been corrupted.

```
.sound
    ASL A                multiply soundbuffer by 8
    ASL A
    ASL A
    ADC #FNlo(sndbuff)  add in address of sound table
    TAX                 put low byte of address in X
    LDY #FNhi(sndbuff) Get hi byte in Y
    BCC nohibyte       If carry then X and Y correct

    INY                 else increment hi byte
.nohibyte
    LDA #&07           OSWORD 7 is sound
    JMP osword
```

*Print strings*

### *ATOM style string - atomstring*

ATOM strings are defined as being groups of characters terminated by a RETURN character (&0D). On entry, X and Y hold the start address in memory of the string to be printed (X holds low byte, and Y holds high byte). A is irrelevant.

A typical call would be:

```
LDX #FNlo (hiscorestring)
LDY #FNhi(hiscorestring)
JSR atomstring
```

On exit, all registers will have been corrupted.

```
.atomstring
    STX  stringptr           Address of string to be printed
    STY  stringptr + 1       is given in X and Y
    LDY  #&00                Y is pointer along the string
.atomstringloop
    LDA  (stringptr),Y       Get next character from string
    JSR  oswrch              print it
    INY                          increment pointer
    CMP  #&0D                is character return
    BNE  atomstringloop     no? go back to start of loop
    RTS                       return
```

### *Microsoft style strings – microsoftstring*

Microsoft strings are defined as being groups of characters, preceded by a byte giving the length of the string. A microsoft string can be set up as follows:

```
.fredstring
EQUB LEN(fred$)
EQU  fred$
```

On entry, X and Y hold the start address of the string (X holds low byte, Y high byte). A is irrelevant.

A typical call might be:

```
LDX #FNlo(fredstring)
LDY #FNhi(fredstring)
JSR microsoftstring
```

On exit, all registers will have been corrupted.

```
.microsoftstring
    STX  stringptr           Address of string to be printed
    STY  stringptr + 1       is given in X and Y
    LDY  #&00                Set pointer to length byte
```

LDA (stringptr),Y	Get length of string
STA len	Save length
.stringloop	
INY	loop counter
LDA (stringptr),Y	get next char from string
JSR oswrch	print it
CPY len	printed all chars?
BNE stringloop	no ? go back to start of loop
RTS	

### *Centre a string - centre*

This routine will centre up microsoft strings on the current line of the cursor. The reason that only microsoft strings can be centred using this routine is that their length is far more accessible than the length of an atom string, although the routine could be adapted for atom strings.

The BASIC equivalent of the routine given below is:

```
PRINT SPC((screenwidth - LEN(A$)) DIV 2);A$
```

On entry, X and Y point to the string to be centred, and A is irrelevant.

A typical call would be:

LDX #FNho(string)	Point to string
LDY #FNhi(string)	
JSR centre	Centre it

On exit, all registers will have been corrupted.

.centre	
STX stringptr	Save Low byte of start address
STY stringptr + 1	Save high byte of start address
LDY #&00	Prepare to get Length byte
LDA #screenwidth	Get screen width
SEC	
SBC (stringptr),Y	Subtract length
LSR A	Divide by 2
TAX	And transfer it to X
LDA #ASC" "	Stand by to print X spaces
.centreloop	
JSR oswrch	Print a space
DEX	Decrement counter
BNE centreloop	and loop until counter is zero
LDX stringptr	Restore string pointers
LDY stringptr+1	
JMP microsoftstring	And print the string

### *Move cursor to X, Y – printtab*

This is a very simple routine to move the text cursor to X, Y. It simulates BASIC's 'PRINT TAB (X, Y)'.

On entry, X and Y hold the X and Y coordinates of the position that the text cursor is to be moved to, A is irrelevant.

A typical call might be:

```
LDX #xcoord
LDY #ycoord
JSR printtab
```

On exit, X and Y will be preserved, and A will have been corrupted.

```
.printtab
    LDA #31          VDU 31 (move textcursor to X,Y)
    JSR oswrch
    TXA             send X coordinate
    JSR oswrch
    TYA             send coordinate
    JMP oswrch      do it and return
```

### *Double height characters - double*

This next routine will only work in Teletext mode, and is thus only suitable for the BBC microcomputer.

The BASIC equivalent of this routine is:

```
DEF PROCdouble(A$)
vpos% = VPOS : pos% = POS
FOR string% = 0 TO 1
PRINT TAB(pos%, vpos% + string%);CHR$&8D; A$;CHR$&8C;
NEXT string%
ENDPROC
```

On entry, X and Y point to the string that is to be printed in double height, A is irrelevant.

A typical call would be:

```
LDX #FNlo(string)
LDY #FNhi(string)
JSR double
```

On exit, all registers will have been corrupted.

double

STX stringptr	Save start address of string
STY stringptr + 1	
LDA #&86	Read text cursor position
JSR osbyte	
STX pos	and store it
STY vpos	
LDX #&02	Print string twice
STX count	
.doubleloop	
LDX pos	Move cursor to X,Y
LDY vpos	
JSR printtab	
LDA #&8D	Teletext code for Double height
JSR oswrch	
LDX stringptr	Restore string start address
LDY stringptr + 1	Either 'microsoft' or 'atom'
JSR string	To centre string JSR centre
LDA #&8C	Teletext Normal height code
JSR oswrch	
NC vpos	Move down a line
DEC count	Done it twice yet
BNE doubleloop	If not then do it again
RTS	Else return

### *Palette handling - ospalette*

This routine performs VDU 19, Y, A, 0, 0, 0. On entry, Y contains the logical colour to be defined, A contains the physical colour to change Y to. (See section 9.4 for details of palette handling.) X is irrelevant.

A typical call might be:

```
LDY #logicalcolour
LDA #physicalcolour
JSR ospalette
```

On exit, Y and X have been preserved, A has been set to zero.

```
.ospalette
PHA          Save physical colour
LDA #19     VDU 19
JSR oswrch
TYA          Get logical colour
JSR oswrch  VDU it
PLA          Get physical colour
JSR oswrch  VDU it
LDA #&00     Pad out with zeroes
JSR oswrch
JSR oswrch
JMP oswrch  and return
```



Another routine for those of you with Electrons or BBC micros with 1.0 or 1.2 Operating Systems is to change the palette with OSWORD 12. This has the advantage of being able to be called from an interrupt routine.

```
.ospalette
    STY paletteblock           Same format as VDU 19
    STA paletteblock
    LDX #FNlo(paletteblock)
    LDY #FNhi(paletteblock)   Note that all registers are corrupted
    LDA #12
    JMP osword
```

This is called in the same manner as before. Note that the 'paletteblock' must contain zeros in the last three locations before the routine is called.

*Wait for flyback - vsync*

The next routine is a must for animation. It will wait for the electron beam inside the VDU (Visual Display Unit) to reach the top of the screen in BBC Microcomputers or the bottom of the screen in Acorn Electrons, and will then return. This is when all shapes should be updated to avoid flickering.

On entry, all registers are irrelevant. A typical call would be:

```
JSR vsync
```

On exit, all registers will have been corrupted.

```
. vsync
    LDA #19   Osbyte 19 is wait for vsync
    JMP osbyte (vertical synchronisation)
```

or

```
. vsync
    LDA #&02
    STA viaier
.vloop
    BIT viaifr
    BEQ vloop
    LDA #&82
    STA viaier
    RTS
```

This routine is only for Issue 0.10 Operating Systems on the BBC micro  
This is at &FE4E

viaifr stands for Versatile Interface Adapter Interrupt flag Register, which is at &FE4D

## 12.2 Analogue to digital routines

### *Analogue to digital value - adval*

This routine reads any A to D channel.

On entry, X holds channel to be read. Y and A are irrelevant.

A typical call might be:

```
LDX #1      Get value of channel
JSR adval
```

On exit, the value is returned in Y and X (Low byte and high byte respectively).

.adval

```
LDY #0      Get ADVAL(X)
LDA #&80
JMP osbyte
```

### *Joystick handler -joystick*

This routine reads either of the two joysticks connected to the A to D converter. Note that the sensitivity of the reading is dependant upon the value of the constant 'joyrange' (0 - insensitive to 127 - very sensitive). The variables 'xcoord' and 'ycoord' are user variables.

On entry, X holds the number of the joystick to be read (1 or 3). A and Y are irrelevant.

A typical call might be:

```
LDX #1      Get readings of first joystick
JSR joystick
```

On exit, all registers will have been corrupted.

.joystick

```
STX temp          Preserve joystick number
JSR adval         Get horizontal reading
LDX temp          Restore joystick number
CPY #joyrange     Is reading within Limit ?
BCS tryleft      See if within other limit
```

.right

```
INC xcoord        Go right
```

.tryleft

```
CPY #256-joyrange Is reading within limit ?
BCC getotherpot  no? try vertical component
```

.left

DEC xcoord	go left
.getotherpot	get vertical component
INX	Get adval (joystick + 1)
STX temp	preserve as before
JSR adval	get reading
LDX temp	restore joystick number
CPY #joystick	all this is as above
.down	except that y coordinate
DEC ycoord	is being adjusted
.tryup	
CPY #256- joyrange	
BCC tryfire	
.up	
INC ycoord	
.tryfire	Get fire button
TXA	Halve X (for fire button
LSR A	mask)
STA temp	(ADVAL (0) AND X DIV 2)
LDX #0	Get ADVAL (0)
JSR adval	
TXA	X holds fire button status
AND temp	AND with mask
BEQ exit	Not held down, then exit
.fire	
JSR firebullet	Else do something exit
.exit	
RTS	Return

### *Oscilloscope*

The program displays the four A to D channels as four different colours (colours 1 to 4). This program will only work on the BBC Microcomputer Model B, as it uses MODE 2, the Analogue to Digital converter, and Hardware scroll. The program also only reads the top eight bits of each channel, as the graphics vertical resolution is only 256. Sampling of the channels takes places every 1/50 of a second.

Some of the ideas in this program can be adapted to other programs. Note the modular construction, with each module as small as possible, so that it could be debugged easily during development.

```

10  REM Oscilloscope V1
20  DIM code 512           set aside area for code
30  PROCassemble         assemble code
40  MODE 2               set up screen mode
50  CALL oscilloscope    call machine code
60  END
70
80  DEF PROCassemble
90  osbyte = &FFF4       set up variables
100 oswrch = &FFEE      constants

```

110	top = &80	zero page location
120	screen = top + 2	
130	temp = screen + 2	
140	DIM colour 3,sidebuffer 255	define vectors
150	!colour = &030C0F30	fill colour vector
160	FOR pass = 0 TO 2 STEP 2	
170	P% = code	
180	[OPT pass	
190	.oscilloscope	entry point
200	JSR clearsidebuffer	
210	JSR readchannels	
220	JSR vsync	
230	JSR scrollsreen	
240	JSR writeside	
250	JMP oscilloscope	
260		
270	.clearsidebuffer	
280	LDA #0	set buffer to 0
290	TAY	buffer is a page long
300	.clearloop	
310	STA sidebuffer,Y	
320	DEY	
330	BNE clearloop	
340	RTS	
350		
360	.readchannels	
370	LDX #4	get four channels
380	.loop JSR adval	
390	LDA colour-1, X	get channel colour
400	STA sidebuffer, Y	put reading in buffer
410	DEX	get next channel
420	BNE Loop	
430	RTS	
440		
450	.vsync	
460	LDA # 19	see section 12.1 for
470	jmp osbyte	OS 0.10 vsync routine
480		
490	.scrollsreen	
500	LDA top	top = 16 bit address
510	STA temp	
520	LDA top+1	store top DIV 8 in the
530	LSR A	6845 CRTC chip, see
540	ROR temp	section 9.3 for
550	LSR A	details of screen
560	ROR temp	scrolling
570	LSR A	
580	ROR temp	
590	LDX #12	register 12 & 13
600	JSR os6845	hold start address of

610	LDX #13	memory to be displayed
620	LDA temp	
630	JSR os6845	
640	# Now increment top	
650	CLC	adjust variable that
660	LDA top	holds start address
670	ADC #8	
680	STA top	
690	LDA top+1	
700	ADC #0	
710	BPL validaddress	
720	SEC	allow for wraparound at
730	SBC #&50	&30000 / &8000 barrier
740	validaddress	
750	STA top+1	
760	RTS	
770		
780	writeside	dump buffer to screen
790	LDX #&FF	start at top of buffer
800	LDY #0	set offset to zero
810	LDA top	add &270 to top
820	CLC	to right hand side of
830	ADC #&70	the screen
840	STA screen	
850	LDA top+1	
860	ADC #2	
870	BPL validaddress2	
880	SEC	again allow for
90	SBC #&50	wraparound
900	validaddress2	
910	STA screen+1	
920	outerloop	
930	LDA sidebuffer,X	get next byte from buffer
940	STA (screen),Y	store to screen
950	DEX	adjust buffer pointer
960	INY	every 8 bytes down
970	CPY #8	the program must add in
980	BNE outerloop	&280 to the address ,in
990	LDY #0	order to move to the next
1000	LDA screen	line
1010	CLC	
1020	ADC #&80	
1030	STA screen	
1040	LDA screen + 1	
1050	ADC #2	
1060	BPL validaddress3	
1070	SEC	
1080	SBC #&50	
1090	validaddress3	
1100	STA screen + 1	

1110 CPX #&FF	buffer all transferred to
1120 BNE outerloop	screen ?
1130 RTS	
1140	
1150.adval	
1160 TXA	preserve channel pointer
1170 PHA	
1180 LDY # 0	get adval(X)
1190 LDA #&80	
1200 JSR osbyte	
1210 PLA	restore channel pointer
1220 TAX	
1230 RTS	
1240	
1250.os6845	
1260 PHA	perform
1270 LDA #23	VDU23;X,A,0;0;0;
1280 JSR oswrch	to put A into 6845
1290 LDA #0	register X
1300 JSR oswrch	
1310 TXA	
1320 JSR oswrch	
1330 PLA	
1340 JSR oswrch	
1350 LDX #6	
1360 LDA #0	
1370.pad	
1380 JSR oswrch	
1390 DEX	
1400 BNE pad	
1410 RTS	
1420]	
1430 NEXT pass	
1440 ENDPROC	

### 12.3 Numerical routines

*MOD – mod*

This routine can be useful when rounding numbers down, or when the remainder of a value is wanted, and the AND instruction cannot be used (e.g. value MOD 3). Note that this method is not one to be recommended for anything other than single byte operations, as the method of repeated subtraction would then be too slow. Also, there is no error checking procedure, so setting Y to 0 would cause the routine to loop forever.

On entry, A holds dividend, and Y holds the divisor. X is irrelevant.

A typical call might be:

LDA #dividend

```
LDY #divisor
JSR mod
```

On exit, A holds the remainder, X and Y are preserved.

```
.mod                Performs A = A MOD Y
    STY temp SEC    Store divisor in temporary location
.modloop
    SBC temp        Set carry ( for subtraction )
    BCS modloop    Repeatedly subtract Y from A
    ADC temp        until A becomes less than zero
    RTS            Add divisor (Note carry clear) and return
```

### *Random number generator – rnd*

A routine which is always useful for games programs is a random-number generator. The following routine generates a pseudo-random number in A. The seed for the random number is three bytes long. This seed can be initialised before using the routine, in order to generate a fixed sequence of numbers. Note that the seed should never contain zero in all three bytes as then the routine will continually give zero.

On entry, all registers are irrelevant. A typical call would be:

```
JSR rnd
```

On exit, A holds the next pseudo-random number. X and Y are preserved.

```
    LDA seed                Get low byte of shift register
    AND #&48
    ADC #&38
    ASL A
    ASL A
    ROL seed +2
    ROL seed +1
    ROL seed
    LDA seed
    RTS
```

## 12.4 Miscellaneous

### *Cyclic redundancy check - CRC*

Cyclic redundancy checks can be useful for error detection when comparing blocks of data. Using the program below you can give any block of memory a 'unique' two-byte signature. Thus you can check that two copies of a program are identical, by seeing if they have the same signature. This method is very secure, as it is very unlikely that two different blocks of memory would give the same signature.

```
0 REM CRC calculator
10
20 signature = &70
30 addr = signature + 2
40 endaddr = addr + 2
50 DIM code 200
60 FOR pass = 0 TO 2 STEP 2
70 P%=code
80 [OPTpass
90.crc
100 LDA # 0                initialise signature
110 STA signature
120 STA signature + 1
130.mainloop
140 JSR crcbyte            get crc for each byte
150 INC addr               16 bit increment
1 60 BNE nohibyte
1 70 INC addr + 1
180.nohibyte
190 LA addr                if at last address
200 CMP endaddr           then end
210 BNE mainloop         else do another byte
220 LDA addr + 1
230 CMP endaddr + 1
240 BNE mainloop
250 RTS
260
270.crcbyte
280 LDY #0
290 LDA (addr),Y          get byte
300 LDX #8                8 bits in a byte
310.loop
320 LSR A                 do crc
330 ROL signature
340 ROL signature + 1
350 BCC nextbit
360 PHA
370 LDA signature
380 EOR # &2D
390 STA signature
```



```

400 PLA
410.nextbit
420 DEX get new bit in byte
430 BNE loop
440 RTS
450]
460 NEXT pass
470 INPUT "start address &"start$
480 laddr = EVAL("&" + start$)
490 INPUT "Length &"length$
500 lendaddr EVAL("&" + length$ + "&" + start$)
510 CALL crc
520PRINT "signature = "&" ; ~!signature AND &FFFF

```

## 12.5 General purpose macros

*Get low byte - FNlo*

DEF FNlo (value) :=value AND &FF An example call is

LDA #FNlo(table)

*Get hi byte – FNhi*

DEF FNhi (value): =(value AND &FF00) DIV &100 An example call is

LDY #FNhi(string)

*Reserve space - FNspace*

DEF FNspace(amount)

P% = P% + amount

0% = 0% + amount

= pass

this line is only relevant on BASIC II or electron

An example call is

.table

OPT FNspace(500)

*16 bit addition – Fnadc*

Provides 16 bit addition.

A typical call might be:

OPT FNadc(&3000, &2000, &2004)

This would add the contents of &2000 (low byte) and &2001 (high byte) to the contents of &3000 (low byte) and &3001 (high byte) and store the result in locations

&2004 and &2005.

```
DEF FNadc(operand1, operand2, result)
[OPT pass
LDA operand1
CLC
ADC operand2
STA result
LDA operand 1+1
ADC operand 1+1
STA result + 1
]
= pass
```

#### *16 bit subtraction - FNsbcb*

Provides ~6 bit subtraction. A typical call would be:

```
OPT FNsbcb(&3000, &2000, &2004)
```

This would subtract the contents of &2000 (low byte) and &2001 (high byte) from the contents of &3000 (low byte) and &3001 (high byte) and store the result in locations &2004 and &2005.

```
DEF FNsbcb(operand1, operand2,result)
[OPT pass
LDA operand1
SEC
SBC operand2
STA result
LDA operand1+1
SBC operand2+1
STA result+1
]
=pass
```

#### *Debugging macro - FNdebug*

This can be inserted anywhere in the sources code to provide an indication of which path the processor has taken through the program. When executed, the routine will make a 'Bleep' sound and wait for a key to be pressed before continuing. All registers are preserved.

A typical call would be

```
OPT FNdebug
```

```
DEF FNdebug
[OPT pass
PHP Save all registers
```

```

PHA
TYA
PHA
TXA
PHA
LDA #7    Make 'Bleep' sound
JSR oswrch
LDX #1    Flush keyboard buffer
LDA #15
JSR osbyte
JSR osrdch    Wait for a key
PLA        Restore all registers
TAX
PLA
TAY
PLA
PLP
]

```

= pass

*16 bit rotation – FNshift*

This provides a 16-bit shift instruction.

A typical call might be

```
OPT FNshift(&2034, TRUE, 2)
```

which would shift locations &2034 and &2035 right twice.

```

DEF FNshift(addr, right, number)
LOCAL shift
FOR shift 1 TO number
IF right [OPT pass :LSR addr +1 :ROR addr :]
ELSE [OPT pass ASL addr: ROL addr + 1: ]
NEXT shift
= pass

```

## 12.5 BASIC routines for use with assembler

*Double height - PROCdouble*

The next two routines can be used to produce double height characters in MODEs 0,1,2,4 and 5. MODEs 3 and 6 (and 7 on the Acorn Electron) have gaps between lines which make it impossible to do double height. To centre the String, type

```
'PRINT TAB ((screenwidth DIV 2) - LEN(A$) DIV 2,VPOS);
```

after 'LOCAL I%'. Note that 'block' is a global array which should be DIMensioned at

the start of the program, using, for example, DIM block 9. Note also that 'char' is the character to be defined, in this case it is always 224. The routine currently prints out the characters as it redefines them, although it is possible to suppress this.

```
DEFPROCdouble(A$)
LOCAL I%
FOR I% = 1 TO LEN(A$)
PROCchar (ASC (MID$ (A$,I% , 1)),224)
NEXT I%
ENDPROC
```

```
DEFPROCchar(C%,char)
LOCAL A%,X%,Y%,J%,I%
?block = C%
A% = 10
X% = FNlo(block)
Y% = FNhi (block)
CALL osword                osword is at &FFF1
FOR J% = 0 TO 1
VDU 23, char
FOR I% = 2 TO 9
VDU block?(J% * 4 + I% DIV 2)
NEXT I%
VDU char,10,8
NEXT J%
VDU 11,11,9
ENDPROC
```

#### *Find string in program - PROCfind*

This next routine will find all occurrences of a specified string in the BASIC program, and print out the line numbers in which the string occurs. In its present form, the routine will not find BASIC keywords (FOR, REPEAT, PROC, etc). To allow for this it will be necessary to store the string as a line of BASIC, which could then be used as the target string in the search. So type PROCfind (\$(PACE + 4)). This will find the string specified in the first line of the program, which should be line 0 to avoid the routine searching for the wrong string. The first line of the routine can now be changed to 'DEFPROCfind', as the parameter is now in line 0. Note that the first IF statement is only needed in BASIC I.

```
DEFPROCfind (A$)
LOCAL Z%,A%
Z% = PAGE
REPEAT A% = Z% + 4
IF LEN ($A%) >= LEN (A$) IF INSTR($A%,A$)
PRINT Z%?1 * 256 + Z%?2 ; Z% = Z% + Z%?3 UNTIL Z%?1 > &7F
ENDPROC
```

## 13. GRAPHICS

This chapter is all about fast shape drawing. Like everything else, shapes such as space invaders are stored as a series of numbers in the computer's memory. This chapter consists of two main sections. The first contains a routine which is designed to be used independent of any other program. It allows a shape to be designed, and produces the relevant numbers representing that shape. The second contains two routines which are meant to be included as part of an assembler program. While the program is running, these routines convert the numbers in the memory back into the original shape and plot this shape on the screen at a stated position. These routines have been highly optimised for fast animation.

Some knowledge of graphics is assumed, up to User Guide level. Those of you without this knowledge are recommended to read the Acornsoft book, Creative Graphics, which gives a clear illustration of the graphics facilities available on the BBC Microcomputer and Acorn Electron.

### 13.1 Shape designer – DESIGN

The BASIC program below can be used to design shapes which can be plotted on the screen using the routines described later in this chapter. Remember that smaller shapes are plotted faster so if you wish to move several shapes on the screen at once it is better not to make them too large. The program is only suitable for a BBC Model B, or Electron.

The keys used by this program are:

0,1..... E,F	Colours 0 to 15
V and H	Vertical and horizontal reflections
L and S	Load and Save shapes
Cursor keys	For moving the cursor around
SY%	Height of 'pixels' in large shape.
shape	Byte vector to hold shape.
cursor X	X co-ordinate of cursor in 'pixels'.
cursor Y	Y co-ordinate of cursor in 'pixels'.
command\$	String containing commands.
	String containing keys for colours.
Colour\$	Colours are specified by pressing key corresponding to Hexadecimal digit. Note state of CAPS LOCK or SHIFT keys is irrelevant.
key\$	Holds key pressed.
length	Holds length of shape data.
C%	Zero if command, else holds colour key plus one.

0 REM Design Program

10

-----  
Top Level. Note that FNinitialise returns the mode  
as MODEs cannot be defined in FNs or PROCs  
-----

20 MODE FNinitialise

30 PROCmainloop

40

-----  
Set up global variables, especially those that are  
relevant to the screen mode selected.  
-----

50 DEF FNinitialise

60 lenshape 300

70 DIM shape lenshape

80 REPEAT

90 INPUT "Mode ( 0 - 2 ) ?"mode

100 UNTIL mode >0 AND mode <=2

110 RESTORE 230

120 FOR I% = 0 TO mode

130 READ pixbits

140 NEXT I%

150 RESTORE 240

160 FOR I% = 0 TO mode

170 READ W%

180 NEXT I%

190 RESTORE 250

200 FOR I% = 0 TO mode

210 READ pixelperbyte

220 NEXT I%

230 DATA 1, 2, 4

240 DATA 2, 4, 8

250 DATA 8, 4, 2

260 INPUT "Width in X-direction:" NX%

270 INPUT "Width in Y-direction:" NY%

280 byteNX% = NX% DIV pixelperbyte

290 byteNY% = NY%-1

300 SX% = (1024 DIV NX%) AND &FFFO

310 SY% = (1024 DIV NY%) AND &FFF8

320 PROCclear

330 cursorX = 0 : cursorY = 0

340 \*FX 4 1

350 command\$ = "VvHhLISs" + CHR\$&88 + CHR\$&89 + CHR\$&8A + CHR\$&8B

360 colour\$ = "001!2""3#4\$5%6&7'8(9)AaBbCcDdEeFf"

370 Length = NX% \* NY% \* pixbits DIV 8

380 = mode

390

-----  
Main loop. Get a key, and then test to see if it is legal. If the key is legal, then

pass it on to the rest of the routine, which then calls the relevant routine(s).

```
-----  
400 DEF PROCmainloop  
410 PROCcursor  
420 REPEAT  
430 REPEAT  
440 key$ = GETS  
450 UNTILINSTR(colour$,key$) OR INSTR(command$,key$)  
460 C% = (INSTR(colour$, key$) + 1) DIV 2  
470 IF C% THEN PROCcolour(C%-1) ELSE ON  
INSTR(command$, key$) GOSUB 670, 670, 780, 780,  
630, 630, 650, 650, 510, 540, 570, 600  
480 UNTIL FALSE  
490 ENDPROC  
500  
-----
```

Handle cursor control keys, making sure that the cursor does not go off the side of the screen.

```
-----  
510 IF cursorX > 0 THEN PRCCdocursor(-1,0)  
520 RETURN  
530  
540 IF cursorX < NX% - 1 THEN PROCdocursor(1,0)  
550 RETURN  
560  
570 IF cursorY > 0 THEN PROCdocursor(0,-1)  
580 RETURN  
590  
600 IF cursorY < NY% - 1 THEN PROCdocursor(0,1)  
610 RETURN  
620  
-----
```

Handle saving/loading of shapes

```
-----  
630 PROCload : VDU 22, mode: PROCshape(FALSE,byteNX%,byteNY%):  
PROCdisplay : RETURN 640  
650 PROCsave : VDU 22, mode:PROCshape(FALSE,byteNX%, byteNY%):  
PROCdisplay : RETURN  
660  
-----
```

Reflect the shape in Y-maximum Y / 2

```
-----  
670 FORI%--0 TO NX%-1  
680 FORJ%=0 TO (NY%-1)DIV2  
690 temp=FNpoint (I%,J%)  
700 PROCplot(I%,J%,FNpoint(I%,NY%-1-J%))  
710 PROCplot(I%,NY%-1-J%,temp)  
720 NEXTJ%  
73C NEXTI%  
740 PROCshape (TRUE,byteNX%,byteNY%)  
-----
```

```

750 PROCdisplay
760 RETURN
770

```

-----  
Reflect the shape in X maximum X / 2  
-----

```

780 FORJ%=0 TO NY%-1
790 F0RI%=0 TO (NX%-1)DIV2
800 temp=FNpoint(I%,J%)
810 PROCplot(I%,J%,FNpoint (NX%-1-I%,J% )
820 PROCplot (NX%-1-I%,J%,temp)
830 NEXTI%
840 NEXTJ%
850 PROCshape (TRUE ,byteNX%,byteNY%)
860 PROCdisplay
870 RETURN
880
890 DEF PROCplot(X,Y,col)
900 GCOL 0,col
910 PLOT 69, X * W% + 1024, Y * 4 + 640
920 ENDPROC
930
940 DEF FNpoint(X,Y)
950 = POINT(X * W% + 1024, Y * 4 + 640)
960

```

-----  
Plot a large square at the cursor position, looking  
at the final size version at the side of the screen  
to find the color  
-----

```

970 DEF PROCsq(X, Y)
980 GCOL 0, FNpoint(X,Y)
990 MOVE X * SX%, Y * SY%
1000 PLOT 0, SX% - 4, 0
1010 PLOT 81, 4 - SX%, SY% - 4
1020 PLOT 81, SX% - 4, 0
1030 ENDPROC
1040

```

-----  
Display whole shape  
-----

```

1050 DEF PROCdisplay
1060 LOCAL X, Y
1070 FOR X = 0 TO NX% - 1
1080     FOR Y = 0 TO NY% - 1
1090         PROCsa(X,Y)
1100     NEXT Y
1110 NEXT X
1120 PROC cursor
1130 ENDPROC

```



1140

-----  
plot the box cursor at the cursor position  
-----

```
1150 DEF PROCcursor
1160 VDU 5
1170 GCOL 3,3
1180 MOVE SX% * (cursorX + 0.25), SY% * (cursorY + 0.25)
1190 PLOT 1, SX% DIV 2 - 4 , 0
1200 PLOT 1 , 0 , SY% DIV 2 - 4
1210 PLOT 1 , 4 - SX% DIV 2 , 0
1220 PLOT 1 , 0 , 4 - SY% DIV 2
1230 ENDPROC
1240
```

-----  
get a shape from the filing system  
-----

```
1250 DEF PROCload
1260 LOCAL I%, channel
1270 PROCshape(TRUE, byteNX%, byte NY%)
1280 channel = FNopen(TRUE)
1290 IF channel = FALSE ENDPROC
1300 PROCclear
1310 FOR I% = 0 TO (byteNX% * NY%)-1
1320 SHAPE?I% = BGET#channel
1330 NEXT I%
1340 CLOSE #channel
1350 ENDPROC
1360
```

-----  
write a shape to the filing system  
-----

```
1370 DEF PROCsave
1380 LOCAL I% , channel
1390 PROCshape (TRUE, byteNX% ,byte NY%)
1400 channel = FNopen(FALSE)
1410 IF channel = FALSE ENDPROC
1420 FOR I% = 0 TO (byteNX%*NY%) -1
1430 BPUT#channel,shape? I%
1440 NEXT I%
1450 CLOSE #channel
1460 ENDPROC
1470
```

-----  
utility used by procload and procsave to open a file  
-----

```
1480 DEF FNopen(in)
1490 LOCAL W$
1500 VDU 22,7
1510 INPUT "file name ?"W$
```

```

1520 IF W$ = "" FALSE
1530 IF in THEN OPENIN(W$) ELSE OPENOUT(W$)
1540

```

-----  
plot point on final size shape and also plot square on main screen  
-----

```

1550 DEF PROCcolour (colour)
1560 PROCcursor
1570 PROCplot(cursorX,cursorY,colour)
1580 PROCsq(cursorX,cursorY)
1590 PROCcursor
1600 ENDPORC
1610

```

-----  
Wipe previous cursor from screen update cursor's X and Y coordinates, and then plot the cursor at new coordinates  
-----

```

1620 DEF PROCdocursor(X,Y)
1630 PROCcursor
1640 cursorX = cursorX + X
1650 cursorY = cursorY + Y
1660 PROCcursor
1670 ENDPROC
1680

```

-----  
clear the array used to hold the shape  
-----

```

1690 DEF PROCclear
1700 LOCAL clear
1710 FOR clear = 0 TO lenshape - 4 STEP 4
1720 clear!shape=0
1730 NEXT clear
1740 ENDPROC
1750

```

-----  
either write final size shape from shape array or put final size shape into shape array  
-----

```

1760 DEFPROCshape(flag, X, Y)
1770 LOCAL I%, J%, tempx, tempy
1780 FOR I% = 0 TO X
1790 FOR J% = 0 TO Y
1800 tempx=I%+1024 DIV 16
1810 tempy=((J%+640 DIV 4)EOR&FF)DIV8
1820 PROCaccess(flag, (tempy*&280+tempx*8+((J%
AND 7)EOR 7)+&3000), Y+1, J%, I%)
1830 NEXT J%
1840 NEXT I%
1850 ENDPROC
1860

```

-----  
Get/put byte from/to final shape.  
-----

```
1870 DEF PROCaccess(flag,addr , Y , J% , I%)
1880 IF flag THEN shape/(I% * Y + ( Y - J% - 1 )) = ?addr ELSE
      ?addr=shape? (I%*Y+ (Y-J%-1) )
1890 ENDPROC
```

Variables Used:

lenshape	Maximum length of shape (in bytes).
mode	Holds graphics modesselected.
I%	generral loop control variable.
J%	general loop control variable.
pixbits	Number of bits per pixel.
pixelperbyte	holds number of pixels per byte.
W%	Width of pixels in graphics co-ordinates
NX%	Width of shape
NY%	height of shape
byteNX%	Width of shape (in bytes)
byteNY%	height of shape (in bytes)
SX%	Width of pixels in large shape
SY%	height of pixels in large shape.
shape	Byte vector to hold shape.
cursorX	X co-ordinate of cursor in pixels.
cursorY	Y co-ordinate of cursor in 'pixels'.
command\$	String containing commands.
colour\$	String containing keys for colours

Colours are specified by pressing key corresponding to Hexadecimal digit. Note state of CAPS LOCK or SHIFT keys is irrelevant.

key\$	Holds key' pressed.
length	Holds length of shape data.
C%	Zero if command, else holds colour key plus one.

### 13.2 Plotting a shape on the screen

To plot a shape on the screen at a specified position it is necessary to have two routines; one to convert the X and Y coordinates to a memory location on the screen, and another routine to plot a shape at that address.

Two routines are given below which perform these tasks. The method chosen can only be used for plotting shapes to a resolution of 80 by 256 in MODEs 0, 1 and 2. The routines work by Exclusive ORing the shape onto the screen. This has two main advantages over other methods (such as writing the shape on the screen or ORing the shape with the screen memory). The first advantage is that the detail under the shape is not lost when the shape is unplotted, the second is that the same routine can be used for both plotting and unplotting.

### *Convert X, Y coordinate to screen address – getaddr*

This routine doesn't write anything to the screen, all that it does is generate an address where a shape might then be written to the screen, or read from the screen. It will generate an address for MODEs 0, 1 and 2, allowing for hardware scroll. The algorithm used is given at the end of the listing so that the code can be adapted for MODEs 4 and 5.

On entry, X holds the X coordinate (0 - 79), Y holds the Y coordinate (0 - 255), and A is irrelevant.

A typical call might be:

```
LDX xcoord
LDY ycoord
JSR getaddr
```

On exit, X will be preserved, Y and A will have been corrupted. The resultant address is left in 'addr' and 'addr+1' (low byte, high byte) which must be in zero page. Other locations used are 'temp' (1 byte) and 'top' (2 bytes). For speed, these locations should also be in zero page.

```
.getaddr
    LDA #&00          set hi byte of address
    STA addr + 1     to zero
    TYA
    EOR #&FF        invert Y coordinate
    PHA
    OPT FN rotateacc(3)  divide Y coordinate by 8
    TAY              and leave in Y
    LSR A            adjust carry for * &280
    STA temp         save Y/16 in temp
    LDA #&00        set bottom byte of address to 0
    ROR A            put carry into top bit
    ADC top          and add in top of screen
    PHP             save carry flag
    STA addr        store result in addr
    TYA             get y/8
    ASL A           double it for top byte
    ADC temp        of addr .add in Y/16
    PLP             restore carry
    ADC top + 1     and add in top of screen
    STA addr +1
    LDA #&00        set temp to zero
    STA temp
    TXA             get x coordinate
    ASL A           perform two byte multiply
    ROL temp        by 8 because of memory
    ASL A           map
    ROL temp
```

ASL A	
ROL temp	
ADC addr	add in rest of result so
STA addr	far, and store it
LDA temp	
ADC addr+1	
BPL ok	check for hardware scroll
SEC	if over 3000 - 8000 boundary
SBC #&50	then correct address
.ok	
STA add r+1	And store it
PLA	Restore inverted Y coord
AND #&07	Get row number in computed
ORA addr	column, and add it in
STA addr	
RTS	Return

Some words of explanation:

The algorithm used to calculate the screen memory address is:

$$\text{addr} = X * 8 + ((Y \text{ EOR } \&\text{FF}) \text{ AND } 7) + \&280 * ((Y \text{ EOR } \&\text{FF}) \text{ DIV } 8) + \text{top}$$

where X and Y are the coordinates. The reason Y is inverted is so that the bottom of the screen is treated as 0, even though the memory map is the other way round. There are 640 bytes per character line on the display, which is &280 in hex. The variable 'top' is normally set to &3000, except when hardware scroll is taking place. In most applications this will be irrelevant, and so 'top' may be set up at the beginning of the program and then forgotten about. The Y AND 7 and Y DIV 8 operations are performed because of the memory map of the screen, DIV 8 is performed to get to the start of the current character cell, and AND 7 to get to the current byte in the character cell:

top	+0	+8	+&10	+&18	+&20	+&28
+0	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+1	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+2	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+3	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+4	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+5	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+6	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+7	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+&280	+-----+	+-----+	+-----+	+-----+	+-----+	+-----
+&281	+-----+	+-----+	+-----+	+-----+	+-----+	+-----

### *Plotting a shape at a given screen coordinate - doplot*

This routine works in conjunction with the above routine for plotting a shape at a given X, Y screen coordinate.

On entry, all registers are irrelevant. The parameters passed are:

counter     - holds number of bytes in the shape  
addr        - holds screen address to put shape  
depth       - holds height of shape  
shape       - start address of shape in memory

('shape' must be in zero page.)

On exit, all registers may have been corrupted, but all parameters will have been preserved.

```
.doplot
    LDY #&00                set shape offset to zero
    LDA addr+1              push screen address onto
    PHA                    stack for later use
    LDA addr
    PHA
    LDA depth              get depth of shape
    STA rowcounter
    LDA addr                put offset in character
    AND #&07                cell into Y
    STA offset
    LDA addr                adjust address accordingly
    AND #&F8                goto top of character cell
    STA addr
    STY temp

.innerloop
    LDY temp
    LDA (shape),Y          get byte from shape
    INY
    STY temp
    LDY offset
    EOR (addr),Y
    STA (addr),Y
    INY                    Y holds offset on screen
    CPY #&08                bottom of character cell
    BEQ block              if so then go down a line

.noblock
    STY offset
    DEC rowcounter
    BNE innerloop

.nextblock
    LDA shape
    CLC
```

ADC depth	
STA shape	
BCC nohi	
INC shape+1	
.nohi	
CLC	go to the top of the next column
PLA	by resetting address to top
ADC #&08	of current column, and
STA addr	moving to next character
PLA	cell
ADC #&00	
BPL nobound1	
SEC	
SBC #&50	
.nobound1	
STA addr+1	
DEC counter	easier to DEC counter in
BNE doplot	two places and test
RTS	
.block	
LDY #&00	go down a line
LDA addr	addr = (addr + &280)
CLC	
ADC #&80	
STA addr	
LDA addr + 1	
ADC #&02	
BPL noboundary	if the contents of addr are
SEC	greater than &8000 then
SBC #&50	subtract &5000
.noboundary	
STA addr + 1	
BNE nolock	always jump

### *Interfacing getaddr and doplot - plotshape*

The number of parameters may be cut down by having a 'front end' attached to the start of the doplot routine which would make interfacing easier.

On entry to 'plotshape' A would hold the number of the shape to be plotted and X and Y would hold the X and Y coordinates at which the shape is to be plotted.

A typical call to 'plotshape' would be:

LDY #0	Get first shape
TYA	
PHA	preserve shape number
LDA xcoord, Y	get x coordinate
TAX	
LDA ycoord, Y	get y coordinate
TAY	

PLA	get shape number back
JSR plotshape	plot shape

On exit from 'plotshape' all registers may have been corrupted, although all internal parameters ('depth', 'counter', etc.) will have been preserved.

.plotshape	
PHA	Save shape number
JSR getaddr	Get address
PLA	Restore shape
TAY	
LDA shapeloadr, Y	Set up parameters
STA shape	This assumes that
LDA shapehiaddr, Y	the User has set
STA shape+1	up the relevent
LDA shapysize ,Y	tables
STA counter	(shape loadr,
LDA shapedepth, Y	shapysize, etc)
STA depth	