ASSEMBLY PROGRAMMING MADE EASY FOR THE BBC MICRO

Century Communications
London
About the Author

Ian Murray is head of computing at Holloway School, and lectures in BASIC on the BBC Micro at the City Lit. He produced software for ILEA's 380Z program, and has a keen interest in promoting the use of micros for secondary education and developing software that introduces new and improved teaching methods.

Acknowledgements

The following all helped with this book and I should like to thank them: Andrew Pusey and Ian Clarke who wrote the longer machine code programs; Jonathan Inglis who drew the cartoons; and the staff, students and friends of Holloway School who were the guinea-pigs for the way the book is presented.
CONTENTS

Introduction

Chapter 1  Page 1
An overview – What needs to be coded – Binary arithmetic – The machine itself – Why an assembler?

Chapter 2  Page 10
Into simple assembly language – The program counter – Music while you list

Chapter 3  Page 25
Where to store machine code – The assembler – How an assembler works – What are labels? – Assembler passes – Summary

Chapter 4  Page 43
The BBC assembler and simple programs – Line numbering – Entering the assembler – The first program – Running the program

Chapter 5  Page 52
Hexadecimal – Converting binary to hexadecimal

Chapter 6  Page 57
The 6502 chip – More use of the assembler – Negative numbers in the computer

Chapter 7  Page 72
Introducing new instructions and the BBC operating system – BBC operating system calls – OSASCI – Summary

Chapter 8  Page 80
More about the 6502 processor – Zero page addressing – Comments

Chapter 9  Page 85
Other BBC operating system calls – OSWRCH – Why MOD and DIV? – OSRDCH – OSBYTE – OSWORD

Chapter 10 Page 97
Assembler maths – Addition – Subtraction – Summary
<table>
<thead>
<tr>
<th>Chapter 11</th>
<th>Page 108</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other addressing modes</td>
<td></td>
</tr>
<tr>
<td>Other 6502 instructions</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 12</th>
<th>Page 118</th>
</tr>
</thead>
<tbody>
<tr>
<td>More advanced features of assemblers</td>
<td></td>
</tr>
<tr>
<td>Macros</td>
<td></td>
</tr>
<tr>
<td>Conditional assembly</td>
<td></td>
</tr>
<tr>
<td>The call instruction</td>
<td></td>
</tr>
<tr>
<td>Saving machine code programs</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 13</th>
<th>Page 125</th>
</tr>
</thead>
<tbody>
<tr>
<td>Longer programs: The Christmas tree program</td>
<td></td>
</tr>
<tr>
<td>The real time clock</td>
<td></td>
</tr>
<tr>
<td>Screen revolve</td>
<td></td>
</tr>
<tr>
<td>A simple game</td>
<td></td>
</tr>
<tr>
<td>Ladder drawing</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Finale</th>
<th>Page 180</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Appendices</th>
<th>Page 182</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Page 196</th>
</tr>
</thead>
</table>
INTRODUCTION

Machine code is far easier to learn than most people believe. Anyone of any age can discover the value of machine code very quickly if the ideas are clearly presented.

This book avoids nearly all maths and mumbo jumbo. Using lots of simple little programs it will soon teach you enough machine code for you to be able to write your own short programs and understand other longer programs.

The book covers just enough computer architecture for you to be able to handle the concepts behind machine code and have a thorough understanding of the BBC assembler.

The programs have been laid out in the proper ‘assembly program’ manner. Too often programs using assembly language in magazines are very badly displayed. The method in this book is expensive on memory, but very clear to read. So once you understand the method, feel free to shorten the programs at your leisure.

I have assumed that you will want to read the book from the beginning to the end, so you will find information about the 6502 chip, for instance, in several places, but use the index if you want to find all the references to it. Similarly you may want to type in the programs before reading what they do. Fine – but don’t forget to come back to the book or you may find yourself lost in words and phrases you don’t understand.

The book is based on my O level and CSE assembly language and machine code classes. Most of my students have long since whizzed past me in machine code. So I hope you join them – soon!

I hope that you will find machine code as much fun as I do, and don’t be put off by the failures that can sometimes occur, because the satisfaction of success is so rewarding.

Good luck!

IAN MURRAY
Head of Computing
Holloway School
London
CHAPTER 1

AN OVERVIEW

The French chat in French. The Germans chat in German. Computers chat in machine code. But unlike humans, computers can make do with a very simple language. In fact, computers need about as many instructions to get them to do things as five-year-old humans.

The computer ‘alphabet’ needs just two letters a ‘1’ and a ‘0’ - which is much less than the twenty six letters of our alphabet or the several thousand Chinese characters. We call the 1 or 0 a Binary digit - or ‘BIT’.

The Computer Alphabet needs just two letters a ‘1’ & a ‘0’.

In English the two letters ‘A’ and ‘D’ can be put together to make words with different meanings, eg ‘Add’, ‘Dad’. In the same way the 1 and 0 of machine code can be combined to have different meanings. ‘110’ and ‘010’ will have different meanings to a computer.

So a computer ‘word’ is made up of bits. 1000110100110011 or 0011010111110000 or 1111111100111111 could all be sensible computer words depending on the computer that you are using.
A computer word has a standard length for any particular computer. In the example above the standard word was sixteen bits long. Sometimes big mainframe computers built by companies such as IBM and ICL have computer words of as many as sixty four bits or even more. The home or school computers settle for easy and short word lengths.

There is one small problem with machine code. Generally it is different for every make of machine. So 1111000101010011 would mean different things when used with different machines.

However, all computers want to say roughly the same things, as we do. We chat about food, the weather and work. Computers chat about adding up, storing things in memory or collecting things from memory. We humans may translate the word ‘travel’ into French, German, Dutch, etc. For different machines we have to translate the word ‘add’ or ‘store’ - but once it is translated, the computer will happily perform.

WHAT NEEDS TO BE CODED

There are generally four things that need to be coded into machine code for the machine to understand:
1. the different instructions the machine has to perform, e.g. add, store, load
2. the addresses in the machine’s memory associated with these instructions, eg store IN LOCATION 3000
3. numbers
4. ordinary letters of the alphabet and punctuation needed to talk back to us.

Usually only the first of these depends on the machine you use - or more exactly, the ‘processor’ that your machine uses. For example the ZX80, ZX81, Spectrum, RML380Z, NASCOM 2 and NASCOM 3 all use the ‘Z80’ processor made by ZILOG. The BBC and PET machines use the ‘6502’ processor.

Unfortunately having similar processors does not mean that machine code on the PET will run on the BBC as there are too many other differences between machines, but at least one does not have to get to know too many different machine codes.

The method of coding the addresses is pretty standard for all machines and simply involves counting in binary through all the available memory addresses for your machine. In some cases this can be several million, but for home and school computers is unlikely to exceed about 65,000.

Numbers need to be coded in two different ways. A simple
binary counting method is needed for the machine to understand, and for humans who sometimes need things such as decimals there has to be a different coding method.

The letters of the alphabet have a standard code which has been in use for many years. This is called the ASCII code (the American Standard Code for Information Interchange) and is used throughout the world - except by some IBM machines which use their own EBCDIC coding system. However all home and school computers use the ASCII coding system to represent the letters of the alphabet in the machine.

With good luck, this means that the letter A sent from one machine will appear as the letter A on the second machine. The full ASCII code is listed in the Appendices but for now the code for the letter A is 01000001.

**BINARY ARITHMETIC**

You need not be a genius at binary mathematics - and you may never (after this book) do another binary sum. But it is useful to be able to check any additions that you expect the computer to have made. 'Debugging' programs is one of the most time-consuming tasks the programmer has to face. I once spent a day looking for a '9' which I had inadvertently typed as a 'P' with disastrous consequences. With assembly and machine language programming I try very hard not to make that silly type of mistake, let alone a maths mistake. Knowing how to check any sum mathematically can be invaluable.
When we store a number in a binary register or store we get our expected line of bits, each of which can be 1 or 0. Each place position of a bit has its own value. The extreme right-hand place has a value of 1 and the value doubles with each place leftwards. If a 1 shows in that place position, we add that value into our number: if we have a 0, we ignore its place value.

The first few place values are as follows:

<table>
<thead>
<tr>
<th>256</th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

so 1 0 0 0 1 0 1 1 1

is a number with value $256 + 16 + 4 + 2 + 1$. This comes to a total of 279. Of course there is a similarity to decimal numbers which also have place values. The first four decimal place values are one, ten, one hundred and one thousand.

Addition is therefore simple. Take the easy sum of:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
+ & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}
= 32 + 4 + 1 \quad (37)
\]

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
+ & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
\hline
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1
\end{array}
= 16 + 2 \quad (18)
\]

Working out all the values confirms the result - and of course 'one plus nought' is 'one', where relevant.

If the sum is a little harder and needs a 'carry' to the next column, we still treat it as an ordinary addition sum, e.g.

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
= 8 + 1 \quad (9)
\]

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
+ & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
= 2 + 1 \quad (3)
\]

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
= 4 + 8 \quad (12)
\]

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & + & 1 & = & 0. & carry & 1 & to & next & column.
\end{array}
\]
Binary Arithmetic

In this example carrying one to the next column gave us another example of ‘1 + 1 = 0, carry 1’.

The only other simple addition rule you may care to work out or simply memorize is that ‘1 + 1 + 1 = 1, carry 1’, eg

\[
\begin{array}{c c c c}
  & 1 & 1 & 1 \\
+ & 1 & 1 & 1 \\
\hline
  & 1 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{c c c c}
  & 4 & + & 2 & + & 1 \\
+ & 4 & + & 2 & + & 1 \\
\hline
  & 8 & + & 4 & + & 2 \\
\end{array}
\]

In this column we have, after adding a carry from the first column, ‘1 + 1 + 1 = 1’.

Large numbers may come in two halves, e.g. memory addresses. In assembly language programming we need occasionally to put the two halves together to get a final address. But even if these addresses seem to make a horrific-looking sum, the adding is as easy as it has been here.

But one word of warning: here we have coded and done sums on simple whole numbers. We call this type of whole number an ‘integer’. The arithmetic we have described is called ‘integer arithmetic’. The situation becomes a little more complicated when dealing with fractions, ‘decimals’ and negative numbers. Some of these you will find later in the book.
THE MACHINE ITSELF

At this stage we need to have an idea of how the machine works. What follows is true of nearly all computers and certainly all that you are likely to come into contact with.

Computers are really quite obvious in design. They are no more than a simple ‘network’ carrying pieces of data between various destinations. Those who have played with toy trains - and there are not many of us left - will quickly get the idea.

We have in our computers an enormous main station

We have in our computers an enormous main station with thousands of platforms (called ‘registers’) where we park pieces of data. These pieces of data can be directed through the signal box (called a ‘control-unit’) to all sorts of destinations including other platforms and outside destinations such as printers, video screens, (called ‘monitors’), tape and disc machines, etc. As with all good railway networks, we may want to shunt around or add to our pieces of data. We do this in the computer’s marshalling yard (called ‘arithmetic logic unit’) and, usually having added to or rearranged the pieces of data, we will direct it to be parked at one of our platforms. Sometimes we will direct it to travel straight from the arithmetic logic unit out to a printer or disc.
Your task as a machine code programmer using assembly language is to take direct control of this railway network of data and make certain that everything gets to the right place in the right order at the right time. As any person who has tried running British Railways will tell you, this is no easy task. But if you have some common sense and don’t panic, you will succeed.

As a quick example of your task, to print the word ‘HELLO’ to the screen, you will have to know which register (railway platform) holds the letter H and direct that to the screen. Then you will have to do the same for the other letters from their registers till you have sent all five letters on their way. Of course you will need to count out each letter as you direct it to the screen till you have sent the fifth letter.

It sounds a bit tedious and more complicated than writing in BASIC, but in the early years of computing in the 1950s and 1960s, this was the normal method of programming. Anyway you have no right to complain as you elected to learn machine code and assembly programming!

The computer industry does not talk about a railway network but about the CPU (see Figure 1). This stands for the Central Processing Unit and consists of the three elements I mentioned above:

1. The ALU (Arithmetic Logic Unit)
2. The Control Unit
3. The Immediate Memory (with its thousands of memory registers).

Connected to the CPU are all the items of machinery that you expect such as VDUs (Visual Display Units), keyboards, printers, tape machines, disk drives, etc. Anything that is connected in this way is called a ‘peripheral’ which only means it is not part of the heart of our railway network of data, but a destination added on to it.

**WHY AN ASSEMBLER?**

We humans find it very difficult to read machine code binary words as if they were a normal language. Quite simply we get lost in that welter of 1s and 0s. Anyway, as I said before, the number of different instructions required to control the machine
is relatively small. What is needed is a very simple translation program that recognises those instructions written in a way that we humans might understand and produces the binary patterns that the machine understands. Such a program is called an 'assembler'.

We describe our programs written for the assembler as 'assembly programs'. Initially they hardly seem readable at all, but just imagine how readable a pile of binary codes would be. Assembly languages are said to be 'low level' languages as they are just one step away from the machine code itself and each assembler instruction can be translated directly to a machine code binary pattern.
This is not true of BASIC, COBOL, Pascal or ALGOL which are described as 'high level' languages because they are designed round the problem-solving needs of the programmer. COBOL, for instance, is a language in which defining records and files is relatively easy. So the 'COmmon Business Orientated Language' is likely to be used by programmers in the commercial world who would never dream of writing file handling programs in assembly language. Another example is the simple BASIC instruction 'PRINT', which places characters on the screen. This actually involves several machine code instructions and will involve several assembly instructions.

Assembly instructions are usually called 'mnemonics' - which means a code word that is just a sort of shorthand for the actual instruction.
CHAPTER 2

INTO SIMPLE ASSEMBLY LANGUAGE

If you think back to the railway network, it may occur to you that in many ways the most important part of the network is the ‘marshalling’ or ‘goods’ yard where the trains are put together, and trucks added to each other and shunted around before being ready for dispatch on the rest of the network.

The same is true of the computer network. The arithmetic logic unit (ALU from now on) could claim to be the most important part of the central processing unit. In many computers, and in particular in microcomputers, little data can travel anywhere without passing through the ALU’s most important register, the ‘accumulator’.

The accumulator register stores the result of any calculation, logical operation or data transfer with a peripheral. (Like all truths this is only mostly true.) The instructions available to you, the programmer, will always include a large selection involving use of the accumulator.

At this stage it is worth appreciating that you would be foolish to reinvent the wheel, or the computer, or any of its software. Most manufacturers (and Acorn is no exception with the BBC Micro) provide you with entry points to the machine code programs that control their microcomputer - and expect you to use their machine code. Thus the considerable task of placing a character on the screen in the right place or even designing a letter of the alphabet to place on the screen has already been done. Don’t waste time doing it from first principles: if everyone had done so, civilisation would still be fiddling with Archimedes’ screw!

Again most manufacturers usually guarantee that your software will run after they have upgraded the machine only if you use the machine code routines they supply. Acorn is no exception, and at least one large software company writing programs for the BBC Micro has learnt this to its own cost. Candidly, except for your own amusement, it is better to be boring and useful in your machine code programming, rather than clever and useless.
Using the accumulator

The following is a short selection of some of the ways that you might like to use the accumulator:
1. Placing a character stored in the accumulator to a defined place on the screen
2. Adding another number to what’s already in the accumulator
3. Comparing a number or character in another register to the one that is stored in the accumulator
4. Reversing - rotating or shifting in some way - the character or number stored in the accumulator

To do any of this actually requires the accumulator to be loaded up with the character or number you select. The instruction to do this, written as a mnemonic, looks like this:

LDA #???

LDA means Load Accumulator
# means actual number follows and not an address
?? is the number you want to place in the accumulator

If you wanted to put the letter A into the accumulator before sending it out to the screen, you would need to load the accumulator with the code number for the letter A. Earlier you will have discovered that this is binary pattern 01000001 - or 65 in decimal.

The full mnemonic to load the accumulator with the letter A is:

LDA #65

This, then, is your first mnemonic. However, you will need to understand the role of the program counter if you are going to include this mnemonic into your first assembly program.

THE PROGRAM COUNTER

The program counter is the name of a special memory location in the central processing unit. Usually it is found in the control unit section of the CPU. As its name suggests, it has the task of counting through the program, making certain that all the instructions are done in the right order or sequence. For that reason it is sometimes called the ‘sequence control register’.

This other name is much more useful in helping you understand precisely what the program counter does. You must remember that your ‘assembly program’ is a sensible and
logical series of instructions to the computer. You will (of course) have written your program so that the order of the instructions is the one that you want. Remembering that these instructions have to be stored somewhere in the computer's memory, you will have stored each instruction one after the other in some safe part of it.

Look at Figure 2: you may have decided that memory location 2000 was a sensible place to store the program. So each step in your program is stored from 2000 upwards. The job of the program counter is to store a number which is the memory location number of the next instruction that has to be done. So after it has done instruction 3 at memory location 2002, the number in the program counter is 2003 which is the number of the next instruction that has to be processed (see Figure 3).
Fig 3 PROGRAM COUNTER (LATER)

After each instruction 1 is added to the program counter so that it is already showing where the next instruction is to be found. If you have ever been boating, you will be familiar with the cry 'Come in Number 7'. You of course obey! The number of your boat is hung on a board so that everyone knows which is the next boat that has to come to shore. In the same way the program counter 'advertises' which is the next instruction to be done.
Sadly, computing is not that simple ... well, not with the small home microcomputers. You remember that they have memory stores that are only eight bits long. That is usually not enough to store all the details for an instruction. This means that we have to use two memory locations one after the other to store the full instruction - or even three or four locations. For example the instruction LDA #65 needs two memory locations: the LDA must be stored in one location and the 65 in the next. If you look at Figure 3 again you will see what happens.

So what does the program counter do?

The designers of the microprocessor chips realised that this could be inconvenient, and so the the correct number is always automatically added on to the program counter. This means that it always has stored in it the memory address of the next instruction, however many memory locations were used for the previous instruction.

In our example LDA #65 is stored in locations 2000 and 2001, and if the computer has finished loading up the accumulator with 65, then the program counter will have 2002 stored in it, because this is where the next instruction starts. (Remember it is a binary pattern that is stored so in memory location 2000 the binary pattern for LDA will be stored. This is 10101001. In 2001 the binary pattern 01000001 for 65 will be stored.)

In the BBC Microcomputer you set the program counter yourself with the correct starting address of your program. You
write in BASIC: LET P% = 2000

or whatever address is suitable.

You now need to know how to store your program in the correct memory addresses. This is very easy as the BBC Microcomputer’s operating system does most of it for you.

**MUSIC WHILE YOU LIST**

Now it is time to try and type in a program. Skip this if you do not yet feel confident. This particular program uses some advanced machine code and assembler techniques and do not be surprised if very little is clear to you at this stage.

Because the memory this program uses conflicts with the memory that the Disc Filing Systems use, you cannot use your discs (if you have them) while using the program. For tape machines, (or disc machines after you have done *TAPE,) there is no problem. Later it will be perfectly easy for you to alter the memory locations this program uses, but for the time being type it in as given here.

The BBC Microcomputer is ‘interrupt driven’. This means that instead of the computer asking each section of the computer in turn, (keyboard, clock, printer etc), if it wants to use the CPU and then doing what the device wants, the computer carries on with its most important task until one of these ‘peripheral’ devices ‘interrupts’ it. It then decides a ‘priority’ for the ‘interrupt’. It may, for instance, carry on with what it was doing, OR it may decide that the device which ‘interrupted’ it was important enough to deserve immediate attention - such as disc access.

You can test this for yourself: load a longish program. List the program. While the program is listing type in:

```
NEW    followed by return.
LIST   followed by return.
```

When your program has finished listing, the two above messages will be on the screen and your program will not list. Although the keyboard had ‘interrupted’ the listing of your program, only the keyboard characters <escape>and <break> have enough ‘priority’ to stop the listing. So all remaining characters (in this case everything that you typed) were stored until the machine was ready to use them - which it did when the program had finished listing.
Other ‘cheaper’ microcomputers do not have this facility fully available. If, while they are listing, you type in NEW and LIST nothing happens because it is not the turn of the keyboard to be looked at by the CPU. The listing must finish BEFORE anything else happens. This can lead to frustrating delays.

The MUSIC WHILE YOU LIST program makes use of the ‘interrupt’ facility of the BBC Micro.

How the program works

There are certain ‘events’ in the BBC Micro which constantly interrupt the CPU. Some of these ‘events’ such as the counter/timer can be switched off from interrupting - or back on again - depending on your need for the ‘event’. One of the events that can be noted - and switched on or off from interrupting - is the start of a ‘pulse’ for the ‘vertical synchronisation’ of your television or monitor. There are 50 of these pulses every second. Usually you, the programmer, are not interested in them and so the ‘event’ of the ‘pulse’ is not allowed to interrupt the CPU. But the pulses do provide a very quick and convenient counter. (A list of all the ‘events’ that will interrupt the CPU and which the programmer can use is given on page 425 of the User Guide under *FX13).

Every time a ‘pulse-event’ interrupts the CPU, the CPU is directed to look at this machine code program. After a small number of ‘pulse-events’ this machine code program decides that it is time to sound a note of its trivial tune. This continues for ever - well nearly. Certainly, while you are listing or developing a program, 50 times a second the CPU is being interrupted. During some of those interrupts a note is sounded and then the computer is expected to continue immediately with what it was doing - eg listing a program. This happens so fast that you will not see the screen flicker.

Detailed description

**Line 180** The machine can get confused about which interrupt it is dealing with - particularly while you are developing an ‘interrupt’ driven program. *FX13,4 turns off the ‘pulse-event’ interrupt. It will be turned on again later.

**Lines 280-290** BASIC is quicker and more convenient for defining envelopes, but it does mean BASIC must be present for this program to work.
Lines 380-460  ‘Pointers’ are needed to parts of memory where details of the sounds to be played are being stored. The area of memory chosen is the defined graphics character area. Because the sound data will go from &C00 upwards in memory, any counters to be used have been placed from &CFF downwards.

Lines 620-650  This dividing up of the music data is required by the sound command in assembly language. See page 461 of the User Guide.

Lines 900-940  Every time a ‘pulse-event’ interrupts the CPU the Counter has one more added to it. The value in the Counter is compared to the Playing speed. When there have been 15 interrupts a note is sounded. If no note is to be sounded, the program returns (RTS) back to what the CPU was previously doing - such as listing a program. This happens so fast you will not notice it.

Lines 960-1010  This keeps track of the number of notes in the tune. If the maximum value of notes is reached - everything is reset back to zero for the dreadful tune to repeat. This happens by jumping forward to 1270.

Lines 1060-1090  This is the machine code SOUND call using the OSWORD machine operating system call.

Lines 1140-1220  After playing a note, it is necessary to get the computer ready to play the next note. This ‘maths’ routine works its way through the table of music data, adding 8 for each note. Unlike BASIC which needs four numbers for each SOUND command, in machine code eight memory locations are needed.

Lines 1270-1320  Once the tune has played, everything is reset to the beginning.

Lines 1440-1450  To use the ‘event’ interrupting, you must place the start address of your machine code program into the memory locations &220 and &221. The low end of the address goes into &220. This address (&220 and &221) is fixed by the
BBC Micro. When there is an 'event', the BBC Micro looks at these addresses to see if there is a machine code program it must do. If there is an address in the memory locations it goes off to the correct part of memory. If no address - the event is ignored.

**Lines 1570-1640** This puts the music data into the correct memory locations using the '?' command in BASIC. Remember that '?' means 'the contents of'.

**Lines 1720-1730** You may get irritated by music while you list. This 'plants' function keys with the *FX call which turn on and off the 'pulse-event'.

The full assembly listing is also provided to help your checking. There can be no loading of programs to music. The machine 'freaks' - but once this program has been typed in you can develop any new program happily to music. The <break>key resets everything and you lose the sound.

If you want to, it is possible to type the machine code directly into the correct locations, eg

```plaintext
?0D00 = &EE
?0D01 = &FA
etc.
```

This is not recommended - you will not learn much from the exercise.

---

**THE PROGRAM**

10 REM **************************************************
20 REM * MUSIC WHILE YOU LIST *
30 REM * *
40 REM * WRITTEN FOR THE BBC-MICRO *
50 REM WITH 1.2 OS ROM *
60 REM BY *
70 REM ANDREW PUSEY *
80 REM AUGUST 83 *
90 REM **************************************************
100 REM
110 REM =----------------------------------------
120 REM disable the interrupt request
130 REM as re-running of the program
140 REM can cause the 'event-handling'
150 REM to get confused
160 REM .........................
170
180 *FX13,4
190
200 REM =----------------------------------------
210 REM because the envelopes are
220 REM defined in BASIC, just saving
230 REM the machine code without the
240 REM BASIC will not let the program
250 REM work
260 REM .........................
270
280 ENVELOPE 1,1,0,0,0,10,20,30,80,-4,-1,-9,126;126
290 ENVELOPE 2,1,0,0,0,10,10,30,120,4,-1,6,126,100
300
310 REM =----------------------------------------
320 REM define the memory locations
330 REM to be used for data and the
340 REM program variables
350 REM .........................
360
370 DIM A(4)
380 Start_of_program = &D00
390 Music_data = &C00
400 Temp_low = &CFF
410 Temp_high = &CFE
420 High_pointer = &CFD
430 Low_pointer = &CFC
440 Counter = &CFA
450 Notes_counter = &CF9
460 OSWORD = &FFF1
470
480 REM =----------------------------------------
490 REM define the constants needed
500 REM in the machine code program
510 REM .........................
Max_number_of_notes = 21
Playing_speed = 15

REM set up the pointers for the
REM SOUND parameter block of the
REM music data
REM

?Low_pointer = Music_data MOD 256
?High_pointer = Music_data DIV 256
?Temp_low = ?Low_pointer
?Temp_high = ?High_pointer
?Counter = 0

REM set up interrupt handling
REM assembly program. The program
REM detects the vertical sync
REM pulse for the VDU as an
REM ‘event’ (*FX14,4) and then
REM executes this program before
REM returning to more ordinary
REM matters such as listing the
REM program. The machine code
REM and data are not stored where
REM BASIC can over-write them.
REM
REM
REM
FOR Pass = 0 TO 3 STEP 3
P%=Start_of_program

OPT Pass

WAIT FOR ‘Playing_speed’ NUMBER OF
INTERUPTS TO OCCUR

INC Counter
LDA Counter
CMP #Playing_speed
BEQ JUMP1
940 RTS
950
960.JUMP1 LDA #00
970 STA Counter
980 INC Notes_counter
990 LDA Notes_counter
1000 CMP #Max_number_of_notes
1010 BEQ JUMP3
1020
1030 / PERFORM SOUND WITH 8 BYTES
1040 / FROM 'XY'
1050
1060 LDY High_pointer
1070 LDX Low_pointer
1080 LDA #07
1090 JSR OSWORD
1100
1110 / INCREMENT POINTER TO TABLE
1120 / OF MUSIC
1130
1140 CLC
1150 LDA Low_pointer
1160 ADC #08
1170 STA Low_pointer
1180 BCS JUMP2
1190 RTS
1200.JUMP2 CLC
1210 INC High_pointer
1220 RTS
1230
1240 / RESET POINTER TO BEGINING
1250 / OF MUSIC TABLE
1260
1270.JUMP3 LDA #00
1280 STA Notes_counter
1290 LDA Temp_low
1300 STA Low_pointer
1310 LDA Temp_high
1320 STA High_pointer
1330 RTS
1340 }
1350 NEXT Pass
1360 REM
1370 REM =======================================
1380 REM set the IRQ event handling
1390 REM 'vector' (see page 465 of
1400 REM the User Guide) to point to
1410 REM this machine code program
1420 REM .........................
1430
1440 ?220=Start__of__program MOD 256
1450 ?221=Start__of__program DIV 256
1460
1470 REM =======================================
1480 REM put music sound data into
1490 REM the area of memory reserved
1500 REM for user defined characters.
1510 REM So you may want to relocate
1520 REM where music-data and program
1530 REM variables are sited.
1540 REM .........................
1550
1560
1570 FOR X = 1 TO (4 * Max__number__of__notes)
1580 READ Sound__data
1590 LET Low__sound__data = Sound__data MOD 256
1600 LET High__sound__data = Sound__data DIV 256
1610 LET ?Music__data = Low__sound__data
1620 LET ?(Music__data+1)= High__sound__data
1630 LET Music__data = Music__data+2
1640 NEXT X
1650
1660 REM =======================================
1670 REM set up function keys '8'
1680 REM and '9' to turn on and off
1690 REM the music while you list.
1700 REM .........................
1710
1720 *KEY 9 *FX 13,4:M
1730 *KEY 8 *FX 14,4:M
1740
1750 REM =======================================
1760 REM sound data for music
1770 REM .........................
1780
1790 DATA 1,1,101,8
1800 DATA 2,1,129,10
1810 DATA 1,0,0,15
1820 DATA 3,1,121,5
1830 DATA 1,1,117,5
1840 DATA 2,1,109,5
1850 DATA 3,2,149,10
1860 DATA 2,0,0,15
1870 DATA 1,1,129,8
1880 DATA 2,1,121,5
1890 DATA 3,1,117,5
1900 DATA 1,1,109,5
1910 DATA 2,2,149,10
1920 DATA 1,0,0,15
1930 DATA 3,1,129,8
1940 DATA 1,1,121,5
1950 DATA 2,1,117,5
1960 DATA 3,1,121,5
1970 DATA 1,1,109,15
1980 DATA 2,0,0,15
1990 DATA -1,-1,-1,-1
2000
2010 PRINT "Press function key 8 to hear music"
2020 PRINT "Press function key 9 to kill music"
2030
2040 END

>RUN
0D00
0D00
0D00 OPT Pass
0D00
0D00 / WAIT FOR 'Playing__speed' NUMBER
0D00 / OF INTERRUPTS TO OCCUR
0D00
0D00 EE FA 0C INC Counter
0D03 AD FA 0C LDA Counter
0D06 C9 OF CMP #Playing__speed
0D08 F0 01 BEQ JUMP1
0D0A 60 RTS
0D0B
0D0B A9 00 JUMP1 LDA #00
0D0D 8D FA 0C STA Counter
0D10 EE F9 0C INC Notes_counter
0D13 AD F9 0C LDA Notes_counter
0D16 C9 15 CMP #Max__number__of__notes
0D18 F0 1C BEQ JUMP3
0D1A
0D1A / PERFORM SOUND WITH 8 BYTES
0D1A / FROM ‘XY’
0D1A
0D1A AC FD 0C LDY High__pointer
0D1D AE FC 0C LDX Low__pointer
0D20 A9 07 LDA #07
0D22 20 F1 FF JSR OSWORD
0D25
0D25 / INCREMENT POINTER TO TABLE
0D25 / OF MUSIC
0D25
0D25 18 CLC
0D26 AD FC 0C LDA Low__pointer
0D29 69 08 ADC #08
0D2B 8D FC 0C STA Low__pointer
0D2E B0 01 BCS JUMP2
0D30 60 RTS
0D31 18 JUMP2 CLC
0D32 EE FD 0C INC High__pointer
0D35 60 RTS
0D36
0D36 / RESET POINTER TO BEGINNING
0D36 / OF MUSIC TABLE
0D36
0D36 A9 00 JUMP3 LDA #00
0D38 8D F9 0C STA Notes_counter
0D3B AD FF 0C LDA Temp__low
0D3E 8D FC 0C STA Low__pointer
0D41 AD FE 0C LDA Temp__high
0D44 8D FD 0C STA High__pointer
0D47 60 RTS
Press function key 8 to hear music
Press function key 9 to kill music
CHAPTER 3

WHERE TO STORE MACHINE CODE

The assembly language on the BBC Microcomputer is written as part of a BASIC program. This is unique to the BBC Microcomputer, as most other assemblers stand alone and have nothing whatever to do with BASIC or any other language.

Obviously the machine code made by the assembler has to be found a safe home somewhere in the computer’s memory. Preferably this home should not be part of or even near the BASIC program itself. If your BASIC program with its assembly program ‘grows’, then it may ‘bump into’ where your machine code would like to rest. If your BASIC program is unlikely ever to change in size, you may ‘get away’ with putting the machine code near BASIC. (Some of the examples in this book work like that.) But best of all is to use the facilities provided for you by the BBC Microcomputer.

You now need to know something about how the memory in the BBC Microcomputer is divided up. You would be foolish to try and put ‘your’ machine code where the designers of the BBC Microcomputer have put ‘their’ machine code.

Make sure your Assembly programs don’t ‘bump into’ your Basic programs.
Fig 4 MEMORY MAP OF THE BBC MICROCOMPUTER

You will find a version of Figure 4 in the user guide on Page 500.
As you can see, the whole of the top half of the memory is just not available to you. Memory locations from 65535 decimal to 49152 decimal are all used by the 'operating system'.
As you can see, the whole of the top half of the memory is just not available to you. Memory locations from 65535 decimal to 49152 decimal are all used by the 'operating system'.

This operating system is responsible for all the ordinary and clever tricks of the BBC Microcomputer such as getting the tape loading and saving to work, defining the red function keys, scanning the keyboard for which key you have pressed and putting it on to the screen, allowing you to change colours on the screen easily and quickly - the list is endless.

Then the BASIC language takes up locations 49151 decimal to 32768 decimal. Remember, BASIC needs all this memory so that it can work out what the equivalent machine code is to the program that you typed in in human readable form. For the
more technical of you, BASIC in the BBC Microcomputer is an 'interpreted language'. This means that, though you may have a 500-line BASIC program, it goes to work converting into machine code only the actual instruction it has to carry out next, leaving all the others unconverted till needed.

We are told by the manufacturers that, starting from the bottom, memory locations 0 to 3583 decimal are also reserved for use by the operating system.

This means that the only memory location available to the BASIC program you write, with the assembly program inside it, the BASIC program and the machine code generated by the assembly program and the screen graphics, is from 3584 decimal to 32767 decimal (on a Model B machine).

Whatever you write in your BASIC program is stored by the operating system upwards from 3584 decimal. If you type the BASIC command:

PRINT TOP

then the top memory address used by your program will appear on the screen. TOP is a word reserved by BASIC for this purpose.

Similarly if you type into the machine:

PRINT LOMEM

that will tell you where BASIC intends to keep the values associated with any of the variables used in your BASIC program. Mostly it will be one different from TOP - but you can change this.

The screen graphics needs a slice of memory and this works down from location 32767 decimal. Where it finishes is shown by the BASIC word HIMEM. In every 'MODE' it is different. HIMEM is not the same in MODE 2 and MODE 7. Type into the machine:

MODE 2
PRINT HIMEM

MODE 7
PRINT HIMEM

You can think of HIMEM as a huge wall across memory. Most of the memory below HIMEM can be used by you in various ways.
Think of HIMEM as a huge wall across memory.

None of the memory above HIMEM can be used by you as it is needed by either the screen, BASIC language or the operating system. But this is a wall with rollers under it. It is possible to roll the HIMEM wall down memory creating empty space behind it and before the start of screen graphics.

Typing in MODE 7 and PRINT HIMEM, in a BBC Model B, will have given you the answer 31744. If you type:

HIMEM = 31000

then you will have created 744 empty memory locations which are free for any machine code that you would like to pop into them. The first one free will be 31001 decimal. You can set the program counter to 31001 by typing:

LET P% = 31001

and then all the machine code you assemble will slip into the memory locations free to it. But beware! If it turns out that you need more than 744 memory locations, the computer will not tell you - unless you ask - and certainly no 'error messages' will appear. So after you have finished assembling your program into machine code you would need to type:

PRINT P%
to find out if it was 31744 decimal - or perhaps more - so intruding into the area for MODE 7 screen graphics.

For the technically minded, now is the time to reveal that strictly speaking P% is not the true program counter. It is a variable used by BASIC and the BBC operating system which usually shows what the program counter has stored in it. But it operates just like the program counter.

---

**So Rule 1 for finding space for machine code is:**
1. Move HIMEM down memory
2. Place machine code between HIMEM and start of screen graphics

**Problems with Rule 1:**
1. You have to make certain that you reserve enough space and do not overwrite screen graphics
2. Any change of MODE in your program will reset HIMEM and probably lead to screen graphics in the new MODE over-writing your machine code
3. As TOP shows where the BASIC program itself is, we could place our machine code in some memory locations above TOP. We can type:

   LET P% = TOP + 1000

   Our machine code would then be placed from 'TOP + 1000' upwards. This may interfere with work space that BASIC needs, which starts at HIMEM and works downwards towards LOMEM. But as a temporary measure in MODE 7, it will work adequately.

---

**So Rule 2 for finding space for machine code is:**
1. Set the program counter to a suitable value above TOP

**Problems with Rule 2:**
1. The location above TOP that you set for the program counter must give BASIC enough room to store its variable values
2. With a large BASIC program in MODE 7 or even a small
BASIC program in MODE 2 you must make sure that your resetting of the program counter does not take you too near HIMEM - or you will not allow yourself enough memory space.

You can of course try to overcome one of these disadvantages by shifting the LOMEM wall upwards and fitting your machine code between TOP and LOMEM. BASIC’s variables are dealt with where LOMEM starts, so you would have unused space by doing this shift, e.g.

LOMEM = LOMEM + 250

This would give you 250 free memory locations. You would then need to type

LET P% = TOP

---

Move LOMEM upwards to a suitable new location

---

Rule 3 for finding machine code space:
1. Move LOMEM upwards to a suitable new location
2. Set the program counter to TOP

Disadvantages of Rule 3:
1. You must give yourself enough space for your machine code, or you will overwrite where BASIC keeps its variable values.
2. You must move LOMEM upwards *before* any reference to any variable. Otherwise that variable will sit where it usually does, just one above TOP. Your machine code would then overwrite it.
3. You may move LOMEM too close to HIMEM causing the BASIC to ‘scramble’.

As you remember from Figure 4 showing how the BBC Microcomputer’s memory is organised, there are interesting areas at the bottom of memory. Just as we can move LOMEM and HIMEM around in memory, so we can move PAGE. PAGE always has the memory location of the beginning of your BASIC program. For tape-based machines this is always 3584 decimal (&E00 hex). For disc-based machines this is 6400 decimal (&1900 hex). For technical reasons any movement of the PAGE wall must be in groups of 256 memory locations.

So we can type:

```
PAGE = PAGE + 512
```

This would give us 512 free memory locations from 3584 decimal upwards.

We could then type:

```
LET P% = 3584 (for tape machines)
LET P% = 6400 (for disc machines)
```

This is a bit messy. It would be better to type:

```
LET P% = PAGE - 512
```

This would mean it would not matter if you were using disc or tape machines.

---

**Rule 4 for finding space for machine code is:**
1. Set PAGE to a new value above its current value. This must be a multiple of 256
2. Set the program counter to the old value of PAGE

**Disadvantages of Rule 4:**
1. Any new BASIC program you now load must be forced to load above the machine code, either by manually resetting PAGE to the suitable value, or by specifying the load address when you do load it
2. If you need 260 memory locations for the machine code, then you will have to take the full 512 memory locations, which will mean wasting empty memory locations. This could be significant in a high resolution graphics program

The BBC Microcomputer's operating system seems (with tape-based machines) to have left memory locations 3328 decimal (&D00 hex) to 3583 decimal (DFF hex) free for machine code use. If this is enough space for you, then you can type:

```plaintext
LET P% = 3328
```

or

```plaintext
LET P% = &D00
```

These mean the same. &D00 is 3328 in the hexadecimal counting system which will be explained in detail later. Some of the 'games' programs on the market hide their machine code at &D00. You may have noticed this.

---

**Rule 5 for finding machine code space:**
1. Set the program counter to 3328 decimal (&D00 hex)

**Disadvantages of Rule 5:**
1. Very little space is available to you for the machine code before you start overwriting the BASIC program beginning at 3584 decimal
2. If you have a disc based system, you will find that the system itself uses some of this memory. Then you may find that the machine code starts doing 'funny' things to the disc drives (such as turning them on), or that any disc load operations are incorrect or that use of the BREAK key
overwrites your machine code. This is often the cause of problems you may have found with copying machine code games over to a disc system. These games probably had machine code sitting at &D00 hex.

Rule 5 can sometimes be extended to start at either 3072 decimal (&C00 hex) or 2816 decimal (&B00 hex). These memory locations also get round the problems with disc based machines. But you can only use them if:
1. Nowhere in your program do you use program (or user) defined characters. These are stored from 3072 decimal (&C00 hex) to 3327 decimal (&CFF hex)
2. Nowhere in your program do you use the red function keys. Your plans for these keys are stored in memory locations 2816 decimal (&B00 hex) to 3071 decimal (&BFF hex)

But, as I said earlier, the BBC Microcomputer does try to make life easy for you. The designers were aware that all these different hiding places for machine code could be annoying to organise. They invented a special BASIC command which allows the BBC operating system to fit in your machine code wherever it can find space, thus taking the worry off your shoulders.

If you type:

LET P% = DIM A% 300

then the program counter is set to start at a 300-long block of memory locations which begins at A%. You do not need to know where the block of memory is, but you can find out by typing:

PRINT A%

This is the most common and useful method of hiding machine code as it does not depend on what MODE you are in or on PAGE, HIMEM or LOMEM.

Rule 6 for finding machine code space:
1. Set aside memory space with the DIM statement
2. Set the program counter to the DIM statement

In this book you will find examples of most of these methods of hiding machine code. Generally if the machine code is to be part of a BASIC program, such as a games or educational program, then Rule 6 is the best method. But if you want to engage in 'software' protection or have a machine code sitting in the machine whatever BASIC program is running, then you will have to choose one of the other rules. Probably it is best to hide short programs at 3328 decimal (&D00 hex).

THE ASSEMBLER

HOW AN ASSEMBLER WORKS

To understand some of the features of the BBC assembler it is useful to know how an assembler actually works. If you are familiar with assemblers or just want to know how to use the BBC assembler, then miss out this section.

The assembler engages in a direct one to one translation of the assembly mnemonic to machine code. For each mnemonic there is a single corresponding machine code. The assembler instruction will always sound or look like the instruction that it
has to represent. For example ADC stands for ADd with Carry. NOP stands for No OPeration. So to the relatively experienced eye, an assembly program immediately takes on some meaning, whatever processor it is written for.

In essence the assembler keeps a complete table of all the possible mnemonics and the machine code instructions equivalent to the mnemonics. When you use a mnemonic the assembler looks through its table and picks out the correct machine code, storing it in the memory location designated for it by the program counter - or by a special instruction showing where to store the machine code generated.

Of course, as you learnt earlier, lots of the mnemonics, such as LDA, have a number which goes with them. The LDA instruction needs something to operate on. In the example we chose we wanted to load the accumulator with the binary code for A, and so the full instruction was:

LDA #65

Generally all assembler instructions are split into two halves: What you want to do and Where or how you want to do it

The technical words for this are operation and operand. Generally only a blank space, or sometimes a comma, separates the operation and its operand. When we write LDA #65, it is obvious that LDA is the operation and #65 the operand. Some assemblers - not the BBC assembler - would accept something like JP,2000 as being meaningful.

We describe the blank space or comma as a delimiter. It tells the assembler you have finished typing in the operation. For that reason 'L D A   # 6 5' will not be understood by the assembler. It will either collapse in a fit of computer neurotics, or more properly will tell you in an error message that you have been a fool.

It is also worth noting here that as the table of operations that the assembler keeps is usually in CAPITAL LETTERS, some 'cheap' assemblers will not allow you to type in instructions in small letters or mixtures of upper and lower case. You will be pleased to know that the BBC Assembler is not 'cheap'.

When out of idle curiosity you have been looking at assembler programs, you will have noticed that generally there are three or four columns of 'writing' associated with the assembler. Each column of writing is described as a 'field', and you now
Some messages can bring on a fit of computer nervous breakdown!

know about the operation and operand fields. The two other fields are the ‘label’ field and the ‘comment’ field. The comment field is self-evident. It is exactly the same as the REM statements in BASIC, but far more important. For all practical purposes, an assembly program without comments is utterly meaningless. Unfortunately, probably one of the reasons why you have ventured into this book is that you have come across ‘meaningless’ assembly programs. The comment field should always be used except in the most trivial of cases.

The general layout of the fields is like this:
Label field  Operation field  Operand field  Comment field

To understand the label field we need to understand a little bit more about the simple tasks that our machine code programs have to perform.

WHAT ARE LABELS

The need for labels will become apparent as we decide how we multiply $5 \times 2$. Before you tell me the answer is 10, you ought to know that the 6502 processor chip in the BBC Microcomputer does not have a multiply instruction. This is true of most eight-bit processor chips, but not of larger processors. So you will have to know in your own mind how you got the answer 10.

Obviously the answer lies in repeated addition. You start off with zero and add two to it five times. This is precisely what you might have to do with an assembler.

You will have to keep a counter which will record the five times that you do the addition. This counter can either count from 0 up to 5, or from 5 down to 0. The latter is more commonly used in assemblers for reasons that will soon be apparent.

Look at the following ‘flowchart’ of what has to be done:
START

Set answer store to zero

Set counter store to five

Add two to the answer store

Take away one from the counter store

Is the counter store equal to zero

NO

YES

END

Fig 5 FLOW CHART
If we transfer the diagram to written instructions corresponding to fields, we might write:

<table>
<thead>
<tr>
<th>Label field</th>
<th>Action field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set answer store to zero</td>
</tr>
<tr>
<td></td>
<td>Set counter store to five</td>
</tr>
<tr>
<td>Repeat</td>
<td>Add two to the answer store</td>
</tr>
<tr>
<td></td>
<td>Take away one from the counter store</td>
</tr>
<tr>
<td></td>
<td>If counter store is not zero, then jump back to 'Repeat'</td>
</tr>
<tr>
<td></td>
<td>End the program</td>
</tr>
</tbody>
</table>

The word 'Repeat' has been chosen by me to indicate where in the action steps we would have to go back to, so as to repeat the addition. My label could just as well have been the word 'Again', 'Bongo' or 'Doggy' - but it is worth choosing a label with some meaning to show what you are doing. (In the above example the action field is a combination of the operation and operand fields.)
A label is used to indicate where in the main program we have to jump back to if we want to repeat a section of the main program. It is also used to show where we may wish to jump forward to if we want to miss out a section of program. You choose whatever label names you like, and the assembler keeps a record of what you have used and where you have used them. Then, when it needs to, it makes the corresponding adjustments in the operation field automatically and you do not need to worry! Do not use the same label twice in different parts of the program. You are being 'ambiguous', but the BBC assembler may fail to spot this and your program will not execute properly, even if it seems to assemble properly.

**ASSEMBLER PASSES**

The action of the assembler going through your assembly program is called making a 'pass'. On the first pass through the assembly program the assembler does all the converting to machine code by looking up the mnemonics in its table of codes. It will also keep a record of where you have used labels, and if it comes across another reference to that label, it will make the right adjustments. Look at the list of 'actions' above in our 'action field'. Each of these actions is an 'instruction' and would have an 'instruction code' to go with it.
Forget for a moment the problem of fitting all parts of an instruction into eight bits. Let us pretend that each instruction fits neatly into one memory location. These memory locations happen to be numbered from 101 to 106.

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Instruction stored in the memory location</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Set answer stop to zero</td>
</tr>
<tr>
<td>102</td>
<td>Set counter to five</td>
</tr>
<tr>
<td>103</td>
<td>Add two to answer</td>
</tr>
<tr>
<td>104</td>
<td>Take one from counter</td>
</tr>
<tr>
<td>105</td>
<td>If counter is positive, then jump back to 103</td>
</tr>
<tr>
<td>106</td>
<td>End program</td>
</tr>
</tbody>
</table>

You will notice that the instruction in memory location 105 now tells you to jump back to the memory location 103 - because that is where the instruction is stored that you want to repeat! It is the assembler that automatically changes the label you have used into the correct memory location that you wish to jump to. The assembler will have stored the memory address corresponding to the label, and when it came across an instruction referring to that label, it will have been able to insert into the instruction 105 the memory address that it had previously stored.

When during the program run the computer does the instruction at 105, the program counter is reset to count forward from 103 again rather than continue to 106 where the program would end. It is very important to understand that it is the program counter that stores the address of where to find the next instruction. If your program tells you to go back and do a section of program again, then automatically the program counter is reset to where you are going back to. Of course, when in our example the counter store is zero, the program counter will move to point to the instruction in 106, and so the program will be able to end.

In this example, the assembler would be able to work out all the memory addresses for each label and put them into the right
places for each instruction in one pass through the assembler. This is because it never comes across a reference to a label in the action field before it has come across that label in the label field. This means it knows what the label refers to in the label field when it later comes across the label in the action field. This is not always true. Look at this example:

<table>
<thead>
<tr>
<th>Label field</th>
<th>Action field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scan keyboard</td>
</tr>
<tr>
<td></td>
<td>If key has been depressed then jump forward to ‘Message’</td>
</tr>
<tr>
<td></td>
<td>Print ‘Bad luck’ to screen</td>
</tr>
<tr>
<td></td>
<td>End the program</td>
</tr>
<tr>
<td>Message</td>
<td>Print ‘Well done’ to screen</td>
</tr>
<tr>
<td></td>
<td>End the program</td>
</tr>
</tbody>
</table>

In this case the label ‘Message’ has been found in the action field before there has been any mention of the label Message in the label field. So when converting the second instruction into machine code the assembler does not know what memory address is being referred to by the label Message.

The assembler makes a note in a special table that it has an ‘unresolved forward address’ and where the problem lies. It then continues in the task of converting the remaining instructions into machine code hoping that it will come across a reference in the label field to the label it has found. If and when it does come across a reference in the label field to this label, in its table of unresolved forward addresses, it makes a note of the memory address to which the label applies - and then makes a second pass through the assembly program clearing up all the unresolved forward addresses.

One of the most common errors in your assembly programs will seem to be having instructions to go to places with no label in the label field saying where they are to be found.

So generally assemblers need to make two passes through your assembly program before all the labels and what they refer to can be cleared up. This is true of the BBC Microcomputer.
assembler, but one of the major differences with the BBC assembler is that you have to force the assembler to make two passes through your assembly program with a BASIC 'for-next' loop. This is not true of most other assemblers which will automatically make two passes through your assembly program unless you tell the assembler to do only one pass.

**SUMMARY**

1. An assembler makes a direct translation of the low-level instructions that you have written in assembler mnemonics and converts them to machine code
2. An assembly program will be written in fields. The four usual fields are the label field, operation field, operand field, comment field
3. The assembler will ignore anything that is written in the comment field
4. Fields are separated by blank spaces though some assemblers also allow commas to separate fields
5. The assembler makes passes through your assembly program and will need to make two passes through the program if you have forward references to labels
6. The usual errors you will make in typing in your assembly program will be a) not typing the operation or operand correctly, and b) being careless over your labels.

**Beware!** Just because your assembly program assembles without any errors does not mean it will work. The 'logic' of your assembly program has to be correct as well as the use of the mnemonics and labels. Machine code 'freaks' will delight in telling you of '10K assembly programs' which, after they have been assembled and executed, simply produce glorious gibberish on the screen. It is very rare that a machine code program will execute (run) the first time after it has been assembled. But you can always hope!
CHAPTER 4

THE BBC ASSEMBLER AND SIMPLE PROGRAMS

The BBC assembler is a remarkable beast. Orthodox users of assembly languages become hysterical at the prospect of using the assembler via BASIC! The Acorn Atom is the only other machine currently available to provide this feature. It does, however, allow a 'small' assembler to have most of the features of a very powerful assembler. Moreover it allows the programmer access to all the BASIC functions, such as SIN and COS for including in an assembly program.

To use this section of the book you will need to be familiar with BASIC.

LINE NUMBERING

As the assembly programs are written using the BASIC interpreter, you have to use line numbers with your assembly program. This is an optional feature on many commercially available assemblers. I suggest that you largely ignore the line numbers and use the AUTO instruction to generate the line numbers automatically.
ENTERING THE ASSEMBLER

The BASIC interpreter knows that you are leaving BASIC and writing an assembly program when - at any stage - you type a right facing square bracket: [. It detects a return to BASIC when you type the left facing square bracket: ].

In MODE 7 these symbols appear as a left-arrow and right-arrow respectively. This is because the teletext character set does not have square brackets.

Before entering the assembler you must set the program counter and decide how you are going to reserve the memory space for your program. This has been dealt with earlier.

THE FIRST PROGRAM

```
10 REM**********************************
20 REM* SIMPLE ASSEMBLY PROGRAM *
30 REM**********************************
40 MODE 7
50 LET Start_of_program =&3000
60 LET Output_letter = ASC("A")
70 LET Screen_start = HIMEM
80 LET Offset = 250
90 REM-----------------------------
100 LET P% = Start_of_program
110 [ OPT 3
120 .Prog LDA #Output_letter
130 STA Screen_start + Offset
140 RTS
150 ]
160 REM-----------------------------
170 INPUT" press return "A$
180 CALL Start_of_program
190 END
```

This program falls into three distinct sections: lines 10-80, 90-160, 170-190.

Firstly notice the wide use of variable names and labels which, even before I explain the program in detail, makes it reasonably readable.

The program simply puts the letter A on to the screen in MODE 7. The program has a similar effect to the command in
BASIC: PRINT TAB(10,6) "A". The program relies on the fact that in MODE 7, there is a single memory location in the computer equivalent to each location on the VDU screen. In such cases we say that the screen is 'memory mapped'. As far as the computer's memory is concerned, the screen is as shown in Figure 6.

This is not directly true of the other MODEs. There is, in the other MODEs, a computer memory location equivalent to each screen location. But the screen location is a 'pixel' or 'dot' of light that you see on the screen and not the space occupied by a whole character. The mathematical way these pixels are arranged is considerably more complex than the MODE 7 memory mapping. If you experiment with the screen addresses in other MODEs, do not be surprised at the odd results you may get!
Lines 10-80: Setting the scene in BASIC

Lines 10-30 Always provide a program heading

Line 50 Because the program is short I know that I can start my program at this address. The ‘&’ in front of the number means that the number is counting in the hexadecimal counting system. This is the ‘worst’ way that I said for finding memory space for the program, but I use it to show that in certain circumstances it is ‘safe’.

Line 60 This stores the ASCII code for the letter A into the variable called Output_letter. Notice that all this section of the program is written in BASIC.

Line 70 Unless HIMEM is changed (moved) by you, it always shows the first computer memory location available to the screen. (In particular it tells the highest memory location available to the BASIC program that you write.) This was explained earlier.

Line 80 As I do not want my letter A to appear in the top left-hand corner of the screen, I add an offset to the start of the screen locations to bring it further down the screen. Refer back to Figure 6.

Lines 90-160 The assembly program

Line 100 This sets the program counter to the selected start of the program.

Line 110 The assembler is entered from BASIC, and immediately a command is given to the assembler to tell it what kind of error listing I want from the assembler. See below for more details.

Line 120 This is the start of the assembly program proper. Notice the three fields of label field, operation field, operand field. In the BBC assembler, all labels must start with a full stop. Strictly speaking this program does not need the label ‘.Prog’ but it is included to show the layout of the program. LDA means Load the Accumulator, and it is being loaded with the ASCII code for the letter A, which is the letter we want output to the screen. Generally all input and output must go via the accumulator.
Line 130 We now STore what is in the accumulator into the computer memory location equivalent to the screen position that we want.

Line 140 This will return the program back to BASIC, after is has printed the letter A.

Line 150 Enter BASIC again and leave the assembler.

LDA means LoaD the Accumulator. So to summarise on these three instructions:
STA means STore the Accumulator somewhere else. A copy of the accumulator is transferred to the specified memory location.
RTS means ReTurn from Subroutine. This will restore the program counter to the value it had before a block of program - such as this assembly program - was executed (run). BASIC also needs the program counter in order to function. By using the RTS, we put back into the program counter the value it had when it was last used by BASIC. So BASIC can continue from where it had got to before the assembly program was executed.
Lines 170-190: Getting the program to run

Line 170 As we do not want everything to happen before we are ready, this allows us to set the process into action ourselves.

Line 180 'CALL' is a BASIC command that tells BASIC to surrender the program counter to a machine code program sitting at the memory location given after the CALL. In this case we have already defined the 'Start_of_program' to be memory location &3000. It is because BASIC has to surrender control of the program counter, that we use the RTS assembler instruction to hand the program counter back to BASIC.

Line 190 Self-evident, but good practice.

The OPT Command

Do not confuse this with the *OPT commands. It is entirely different.

The OPT command is an instruction to the assembler to provide you with information about how the assembling of the machine code has progressed. It is known as a 'pseudo-operation' or 'assembler directive'. This is because no machine code is generated by the assembler when it meets the OPT command.
OPT 0  No information provided on screen about your errors.  
     No assembler listing.

OPT 1  No information provided on screen about your errors.  
     Your assembly program is listed.

OPT 2  All your errors are reported. But no full assembler 
     listing is provided.

OPT 3  All assembly errors are reported. A full assembly 
     listing is provided.

Generally you will want OPT 3. You may want OPT 2 when you 
have a long assembly program and you only want to see the 
errors on screen. It can be tedious to watch a whole assembly 
program being listed. Supreme confidence would be a feature 
of OPT 0.

**RUNNING THE PROGRAM**

After you have typed in the program and run it you will see:

```
3000   OPT 3
3000 A9 41  .Prog LDA #Output__letter
3002 8D FA 7C  STA Screen__start + Offset
3005 60   RTS
press return
```

*Knitwear for the young professor of machine code!*
This is the listing produced as specified by OPT 3. As the program is correct, no errors have been reported.

You will notice that the machine code has not been produced for you to view as a binary pattern. It is assumed that even the 'professors' of machine code find it difficult to read machine code, and so the hexadecimal counting system equivalent of the binary pattern has been produced.

The first column of numbers contains the addresses in hexadecimal where the assembler has placed the machine code. As we specified that the start of the program was to be &3000, (the '&' means a hexadecimal number follows), the assembler has performed correctly.

Because OPT 3 is an assembler directive, no machine code is generated when the assembler finds OPT 3. It does however acknowledge its existence.

The binary pattern at each memory location is shown below, with its hexadecimal and decimal equivalent.

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
<th>Hex</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>10101001</td>
<td>A9</td>
<td>169</td>
</tr>
<tr>
<td>3001</td>
<td>00100001</td>
<td>41</td>
<td>65</td>
</tr>
<tr>
<td>3002</td>
<td>10001101</td>
<td>8D</td>
<td>141</td>
</tr>
<tr>
<td>3003</td>
<td>11111010</td>
<td>FA</td>
<td>250</td>
</tr>
<tr>
<td>3004</td>
<td>01111100</td>
<td>7C</td>
<td>124</td>
</tr>
<tr>
<td>3005</td>
<td>01100000</td>
<td>60</td>
<td>96</td>
</tr>
</tbody>
</table>

You will have noticed that the assembler does not list separately every memory location used to store the machine code, you will remember that the eight-bit length of each memory location is not enough to store the full details of some instructions. The assembler has listed the memory location where the start of each instruction has been stored.

The assembler has stored the instructions thus:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>LDA</td>
</tr>
<tr>
<td>3001</td>
<td>65 decimal (the letter A)</td>
</tr>
<tr>
<td>3002</td>
<td>STA</td>
</tr>
<tr>
<td>3003</td>
<td>low half of screen memory address</td>
</tr>
<tr>
<td>3004</td>
<td>high half of screen memory address</td>
</tr>
<tr>
<td>3005</td>
<td>RTS</td>
</tr>
</tbody>
</table>
The contents of 3003 and 3004 need explaining. We have seen that the largest number we can represent in an eight-bit memory address is 255 decimal. You will also remember, by looking at the screen map in Figure 6 that the MODE 7 screen starts at 31744 decimal. For this reason the screen needs the contents of two memory addresses - or sixteen bits - to represent such addresses. In the hexadecimal system 31744 is written as &7C00. Our program added an offset of 250 decimal to the start of the screen. This is &FA in hexadecimal. So the full screen address where our letter A will appear is &7CFA.

For technical reasons concerning the design of an eight-bit microprocessor, any memory address is stored 'back-to-front'. The FA is stored before the 7C.

Finally you will not perhaps have noticed that there is no BASIC for-next loop making the assembler pass through the assembly program twice. This is because one pass is sufficient in this case. There are no jumps through the program likely to lead to unresolved labels.

If you now 'press return' as instructed, the letter A will appear a little below the end of the listing sitting on the screen.

You can check for yourself that the assembled program is sitting at memory locations &3000 to &3005. Type into the computer directly:

FOR X=0 TO 5:P.?(&3000+X)"":NEXT

You will see the decimal values above appear. The '?' means in BBC BASIC 'the contents of'.

If you want to see the code in hexadecimal, then type into the machine directly:

FOR X=0 TO 5:P.~?(&3000+X)"":NEXT

The machine code will stay at &3000 until it is overwritten by a BASIC program, etc. You can prove this. Type into the machine:

NEW
LIST
CLS
CALL &3000

The letter A appears in the same place again. However, if you do not clear the screen, and the screen has done some scrolling
the letter A will not appear in the same place. This is a deliberate design feature of the BBC Microcomputer. The BBC Microcomputer has 'hardware screen scrolling', which means that it can update the graphics screen very fast. It does have the 'side-effect' that, after the screen has scrolled, &7C00 is no longer the start of the screen in MODE 7. There is a special register which we can look at to discover where the current start of the screen has got to: all is not lost.
CHAPTER 5

HEXADECIMAL

Increasing references have been made to hexadecimal. It is not essential that you understand the hexadecimal counting system, but you will find working out memory locations, and what fills them, much easier if you know a little hex. Don’t get hexed by HEX.

Don’t get hexed by HEX!

However, dread does fill the hearts of some faced by counting in anything except tens. But if we had had eight fingers on each hand we would have counted in sixteens. In many ways, the decimal counting systems is very ‘inefficient’. The hexadecimal counting system is very convenient for the computer industry because binary numbers divide into hexadecimal easily.

Unfortunately, because we don’t all have eight fingers, we have not got a special ‘symbol’ for ten. We write ten as 10. In fact we haven’t got special symbols for any of the numbers between ten and fifteen. In a burst of boredom it was decided to use the letters A, B, C, D, E and F for the numbers ten to fifteen.

Look at the drawing of the two hands in Figure 7.
Fig 7 HEXADECIMAL FINGERS

The 10 means that I have got one sixteen and no units - which is how our fingers are used in decimal. In decimal 10 means I have got one ten and no units.

So any two-digit hexadecimal number tells you how many sixteens you have, and how many units. Using the following conversion table, any hexadecimal numbers can easily be worked out:

```
1 2 3 4 5 6 7 8 9 A B C D E F   hex
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15   decimal
```

Generally, to show that a hexadecimal number is being used, the ‘&’ sign is placed before the number. Most assemblers - not the BBC assembler - accept an ‘H’ after the number, such as ‘3000H’, to show that it is a hexadecimal number. Some assemblers will not accept a hexadecimal number at all unless the hexadecimal number starts with a decimal digit! These assemblers are a little inconvenient to use but they want the number to be written like ‘0AF34H’ rather than just ‘AF34H’.

However, the BBC assembler just wants the & sign before all numbers. So:

\&35 means three 16s and five units: \( (3\times16) + 5 = 48 + 5 = 53 \)
\&40 means four 16s and no units: \( (4\times16) + 0 = 64 + 0 = 64 \)
\&A1 means ten 16s and one unit: \( (10\times16) + 1 = 160 + 1 = 161 \)
\&5A means five 16s and ten units: \( (5\times16) + 10 = 80 + 10 = 90 \)
Try to work the following numbers out for yourself - a little practice helps!

\[
\begin{align*}
\&85 &= \\
\&B8 &= \\
\&3B &= \\
\&AB &= \\
\&FC &= \\
\end{align*}
\]

In decimal each column of the number has a place value, and the same is true of hexadecimal:

\[
\begin{array}{cccc}
(10*10*10) & (10*10) & (10) & \text{(units)} \\
\text{thousands} & \text{hundreds} & \text{tens} & \text{ones} \\
3 & 6 & 5 & 1 \\
\end{array}
\]

In decimal this is three thousand, six hundred and fifty one.

In hexadecimal the place values follow a similar pattern:

\[
\begin{array}{ccc}
(16*16*16) & (16*16) & (16) & \text{(units)} \\
4096 & 256 & 16 & \text{ones} \\
3 & 6 & 5 & 1 \\
\end{array}
\]

So the decimal value of the hex number \&3651 is:

\[
(3*4096) + (6*256) + (5*16) + 1
\]

This totals 13,905 in decimal.

You will not want to do all these conversions ‘manually’. What is a computer for? So BBC BASIC does it for you:

```
PRINT \&3651
```

will convert the hexadecimal number to decimal and print 13,905 as the answer. To convert a decimal number to hexadecimal type:

```
PRINT ~13905
```

and you will see the answer 3651. Notice that BASIC does not put the \& sign in front of the answer for you. It assumes you know what you are doing. In MODE 7 the ‘~’ sign is printed as a divide sign.
CONVERTING BINARY TO HEXADECIMAL

This is very easy! There is a little trick to learn. If you are intrigued, try to work out why it succeeds.

Take any row of binary digits, such as 10011100. Then divide them up into groups of four under their place values, as follows:

\[
\begin{array}{cccc}
8 & 4 & 2 & 1 \\
1 & 0 & 0 & 1 \\
\end{array}
\begin{array}{cccc}
8 & 4 & 2 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

Any four digits working from the right will convert to a single hexadecimal digit. The decimal value of the right-hand four digits is: \(8+4+0+0 = 12\). 12 is \(\text{C}\) in hex. The next four digits are \(8+0+0+1 = 9\). 9 is also 9 in hex. So 10011100 is \&9C.

This will work for very long rows of binary digits:

1111000010101100

Split it up:

1111 0000 1010 1100

1100 = 8+4+0+0 = 12 = C
1010 = 8+2+0+0 = 10 = A
0000 = 0+0+0+0 = 0 = 0
1111 = 8+4+2+1 = 15 = F

So the equivalent hex digits are:

1111 0000 1010 1100
F 0 A C

So: 11100010101100 = \&F0AC

The conversion in reverse is equally easy: What is \&3F in binary?

<table>
<thead>
<tr>
<th>hex</th>
<th>decimal</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>1111</td>
</tr>
</tbody>
</table>

So \&3F = 11110011.

You should notice that the convenience of hex is that a maximum of two hex digits will provide all the possible values
for an 8-bit register, and a 16-bit address can always be fitted into a 4-digit hex number. So addresses are nearly always written in hex, padding out the number with noughts as required: eg &0A10 or &0004.

One final warning: the BBC assembler, like most assemblers, expects hexadecimal to be written in CAPITAL letters.
CHAPTER 6

THE 6502 CHIP

The 6502 processor has been around since about 1973. Its design, compared to many modern processors, is very simple. So it has a very small number of instructions controlling what is happening in the processor. Figure 8 is an outline block diagram of the processor.

Each of the blocks represents a register inside the 6502 processor where we can store eight bits of data. Discount the program counter, which needs two registers for the high and low halves of the address of the next instruction. You are left with the 'stack pointer', which I will describe below, the accumulator, the P register, and the X and Y registers.

Of course you have available to you for general storage all the registers in the memory chips, and these are connected to the 6502 chip via an address bus and a data bus. The bus is just a bundle of wires connecting the memory chips to the 6502 chip - and it is called a bus because the wires carry information (not people).

The accumulator

The accumulator is a special register in the arithmetic logic unit (ALU) part of the central processing unit (CPU) which holds either the result of a computation in the ALU or a transfer of data in or out of the CPU. If you want to do any 'maths' in the 6502 chip, it is done via the accumulator. If you want your computer to talk to another computer, it is done via the accumulator.

The X and Y registers

These are general purpose registers which you can use to park eight bits of data temporarily. You can 'increment' these registers - which means add one to them. You can store data from these registers in general memory, transfer data from these registers to the accumulator or even use the registers for a kind of indexing of memory.
DATA "BUS" because the wires "carry" data

1 STACK POINTER

(high) PROGRAM
(low) COUNTER

'X' REGISTER

'Y' REGISTER

ACCUMULATOR

PROCESSOR STATUS 'P'

ADDRESS "BUS" because these wires carry the addresses of the memory locations

Control UNIT which you cannot access directly

To the memory chips

To the memory chips
The P register

The P stands for Processor status. You will shortly discover that if you do not have any idea what is going on in the chip, however powerful it may be, then you are 'lost'. Another word to describe the processor status register is the flags register. This is because each of the eight bits in this register can be used to indicate or 'flag' that something has happened inside the processor.

The eight bits of the processor status register have special significance as follows:

<table>
<thead>
<tr>
<th>Bit number:</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Break</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zero</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Carry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The sign flag If at any time the accumulator has a negative number in it, then you will find that 'bit 7' of the P register will have a 1 in it - otherwise it will have a 0 in it.

The overflow flag If at any time a calculation in the 'two’s complement' system of arithmetic in the ALU results in a number that is too large to be stored in the accumulator and might be incorrect, then the 'bit 6' of the P register will be 1: otherwise it will have a 0 in it.

The break flag It is possible to break into a machine code program. This is called 'interrupting'. If the interrupt has been caused by the assembly BRK instruction, then 'bit 4' of the P register will be 1 - otherwise it will be 0.

The decimal flag It is possible to operate the processor in a special way for decimal addition. This maths system is called 'Binary Coded Decimal' or 'BCD' for short. This is because it can be time-consuming to convert every calculation you make from binary to decimal. If the processor is operating in BCD, then 'bit 3' of the P register will be 1: otherwise it will be 0.

The interrupt flag Sometimes you will not want the processor to be interrupted for any reason. If 'bit 2' of the P register has a 1 in it, then the processor will ignore all interrupts. The BBC Microcomputer uses interrupts to manage all input and output
to the processor - even from the keyboard. One effect of setting bit 2 of the P register in the BBC Micro would be to disable the keyboard! This is because an interrupt signal from the keyboard would be ignored.

**The zero flag** This is the most frequently used flag in the P register. Any instruction to the processor which results in any of the internal registers having zero stored into it will result in ‘bit 1’ having a 1 stored in it. Note that if the result of an instruction sets a register to zero, then a 1 (not a 0) is flagged in bit 1 of the P register. This has caused a lot of confusion to beginners. It is because the P register flags results, and a 1 means wave the flag - and a 0 means don’t wave the flag.

**The carry flag** The carry flag is set to 1 in ‘bit 0’ of the P register if the result of a computation in the ALU would have led to a 1 being carried into another register - if it could. You have to decide what to do with this carry as part of your programming. The carry flag is also used in a different way with the ‘shift’ and ‘rotate’ instructions. After the zero flag, the carry flag is the next most used flag in the P register.

**The stack pointer**

Nearly all computers need a part of the main memory that can be used for temporary working storage. You will appreciate that the 6502 chip with so few ‘internal’ registers has little space for ‘working storage’. So the 6502 chip sets aside memory locations 256 to 511 of your memory chips for your working storage.

This working storage behaves in exactly the same fashion as a human stacking a pile of books.

**Adding to the stack** To start, all the possible memory locations from 256 to 511 will be free. The ‘stack pointer’ will have 256 stored in it, as this is where the first free storage place is. Then you decide to stack your first book on the table. Memory location 256 is now used up, and so the first free memory location in the stack is at memory location 257. Six more books are piled on to your pile or stack using memory locations up to 262. Now the stack pointer shows 263 as the next free memory location.
Taking away from a stack If you have a pile of books, then the only book that you can remove, without threatening to upset the whole pile, is the one on top of the pile. This was of course the last book that you will have added to the pile. The same is true of the stack in memory. The only memory location that you can remove from the stack is the last one you put on it. So if you want the book three down in the pile, then you have to take away the top two books and either throw or give them away. Then you can get hold of the book that you really want. This is also true of the stack in memory. If you want something that is slotted in the middle of your stack, you must first get rid of the items on the top of the stack before you can retrieve your particular one.

This is called a 'Last In First Out' organisation of memory - 'LIFO' for short. The stack pointer always shows the next free position memory location in the stack. There are several 6502 instructions that allow you to add or take away from the stack.

The 6502 is different from many processors in only allowing you memory locations from 256 to 511 for your stack. You will discover that other processors give you the freedom to start your stack anywhere in memory. Some of you may have noticed that to represent the numbers 256 to 511 needs nine bits. I said that all the 6502 internal registers were eight bits. The ninth bit is always set automatically to 1 in the design of the 6502 processor, so that you only ever have to worry about the other
eight bits of the stack pointer register. This means that the lowest address free for the stack is $100000000$ and the highest is $111111111$. This is a 'strange' feature of the 6502 chip.

MORE USE OF THE ASSEMBLER

The following programs are developments of the simple program to place the letter A on the screen. But we shall make a few subtle changes to introduce new instructions.

```
10 REM*************************************************************************
20 REM * Another simple program *
30 REM*************************************************************************
40 MODE 7
50 DIM Start_of_program% 15
60 LET Output_letter = ASC("A")
70 LET Screen_start = HIMEM
80 LET Offset = 100
90 REM ------------------------
100 LET P% = Start_of_program%
110 [ OPT 3
120 .Prog LDA #Output_letter
130 LDX #Offset
140 STA Screen_start,X
150 RTS
160 ]
170 REM ------------------------
180 INPUT "Press return "A$
190 CALL Start_of_program%
200 END
```

You will notice one small difference between this and the previous program. We have introduced the X register to hold the ‘offset’ that we want. In fact we have used the X register to ‘index’ which memory location we want - once we have decided where the screen started. Notice that the LDX command needs the ‘#’ sign if we want to put an ‘immediate’ number into it.

The word immediate means that an actual integer number is put into a register. The values range from 0 to 255 (&0 to &FF). The registers you can put an immediate number into are the X, Y and accumulator registers. Not suprisingly LDY #?? will put an immediate number into the Y register.
Indexing means that we use the X or Y registers to indicate how far from the base address we wish to access. If, as in Figure 9, the base address is &1000 and the index in the X register is &03, then the final address accessed is &1003 (note that all the addresses are in hexadecimal). As the next program will show, this becomes useful when we wish to access a table of addresses.

**Fig 9 INDEXING**

When you run this program you will see that in the second line of the listing the machine code for ‘LDX #' has been produced and the offset after it is '64'. This is the hexadecimal value of 100 decimal: \((6\times16)+4\).

You will also notice that the machine code for the 'STA' command has changed from '8D' in program one to '9D' in this program. This is because the STA instruction now expects an index.

In the first program the STA was referring to an absolute address. The words absolute address mean exact address as printed. In the first program we put the letter A at the exact address which was 'Screen_start + Offset'. Note that the maths here has been done for us, and the exact address '&7CFA' was generated as machine code by the assembler.
In this program the 6502 chip uses the X register to add on to our base address when we execute (run) the program. So the machine code in the third line of the listing is:
9D 00 7C
You will remember that the addresses are put in ‘back to front’. So the address is &7C00. The &7C00 is the value of HIMEM in MODE 7, the start of the screen. This is the base address. So it looks as if we are going to put our letter A at &7C00. But the X register holds the index from this base, and when you press the return key as instructed, the letter A appears in exactly the same place as last time!

Finally notice that, because we used the DIM statement to allocate space for our machine code, it has not assembled at the address &3000 we specified last time. The address at which it assembles will be different on tape-based machines and machines with disk interfaces. It will also be different if you have made even slight changes in your typing from what I have written.

You may wonder, ‘What is the point of this complication?’

It allows us an easy method of drawing a line of As across the screen - or any other character for that matter.

```
10 REM ********************************
20 REM * Better Assembly Program    *
30 REM ********************************
40 MODE 7
50 LET DIM Start_of_program% 15
60 LET Output_letter = ASC("A")
70 LET Screen_start = HIMEM
80 LET Offset = 100
90 REM ----------------------------
100 LET P% = Start_of_program%
110 [ OPT 3
120 .Prog LDA #Output_letter
130      LDX #Offset
140 .More STA Screen_start,X
150     DEX
160     BNE More
170     RTS
180 ]
190 INPUT "Press return "A$
200 CALL Start_of_program%
210 END
```
Type this in and RUN it, but do not press Return yet.
You will notice that two more instructions have been added. ‘DEX’ means DEcrement the X register, ie take one away from what is the X register. ‘BNE’ means Branch Not Equal to zero. The instruction forces a jump back in the program if the X, Y or A registers do not have a zero in them. In this case we are solely interested in what happens to the X register. The accumulator does not have zero in it because we have put &41 (65 decimal) into it.

When you run this the machine code generated will be:

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine code</th>
<th>Listing</th>
</tr>
</thead>
<tbody>
<tr>
<td>A9 41</td>
<td></td>
<td>OPT 3</td>
</tr>
<tr>
<td>A2 64</td>
<td></td>
<td>.Prog LDA #Output__letter</td>
</tr>
<tr>
<td>9D 00 7C</td>
<td></td>
<td>LDX #Offset</td>
</tr>
<tr>
<td>CA</td>
<td></td>
<td>.More STA Screen__start,X</td>
</tr>
<tr>
<td>D0 FA</td>
<td>BNE More</td>
<td>DEX</td>
</tr>
<tr>
<td>60</td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

I have left the address field blank as this will vary according to your typing in, disk- or tape-based machines.
You will notice that taking one from the X register uses only one ‘byte’ of machine code. It is always useful to total up the number of bytes of memory location that you have used or expect to use, as there are times when you will not DIM enough space, with bizarre results. Machine code ‘freaks’ are always trying to take fewer and fewer bytes to perform the same task. This is usually just a ‘game’, as there are only a few specialised times when lopping a few milliseconds off the time that your program takes to execute is important.

The ‘BNE More’ has produced interesting machine code. The machine code for the BNE is ‘&D0’. But notice that the ‘More’ has been converted to ‘&FA.’

The branch instruction has told the 6502 processor to jump back to where the label ‘More’ is and reset the program counter accordingly - as long as no register has a zero in it causing the zero flag in the P register to be set to a 1.

One might expect the address of the start of the instruction ‘STA Screen_start,X’ to be placed after the ‘&D0’ so the computer would know which address to jump back to. This is the case in many processors and is what happens when one uses the ‘JMP’ instruction with the 6502 chip. There is, however, a cleverer way for the 6502 processor to handle the branch type of jump.

Instead of jumping forwards or back to an exact ‘absolute’ address, the branch instruction tells the processor to move forwards or backwards in memory a certain number of bytes from the current value of the program counter. It tells the 6502 chip to move forwards or backwards ‘relative’ to the current value of the program counter. This is called ‘relative addressing’. It has two advantages over absolute addressing.

**Advantages of Relative Addressing**

1. The machine code that is generated does not depend any longer on an exact or absolute address and so can be located anywhere in memory. The machine code written in this way is called relocatable code. It is good practice to try and write relocatable code because the code can be transferred easily from machine to machine or enhanced later without your having to worry about particular memory addresses.

2. Only two bytes are needed for the instruction, which is one less than an exact jump to an absolute address. If space is at a premium, this will be significant.

Finally you will notice that, though I have used a label (.More)
in this program, I have still only made one pass through the assembler. This is because the label does not involve any forward referencing.

But why, you ask, is the ‘More’ translated as ‘&FA’? The secret lies in how the computer handles negative numbers. Before we deal with the negative numbers, press Return and watch the line of As appear almost instantaneously. The As should start at the top left of the screen and finish where the original A was. I am sorry that your listing was overwritten, but your program itself is still safe!

NEGATIVE NUMBERS IN THE COMPUTER

Till now we have assumed that the binary pattern for eight bits will range from 0 to 255 decimal, which is 11111111 in binary. However this does not allow any minus number to be represented because all the eight bits are being used to represent the positive numbers 0 to 255. No bit is available to signal a minus number.

Obviously, as we have discovered with relative addressing, we will want minus numbers. The extreme left-hand bit in any memory register may be used to signal a minus number by having a 1 put into it. In effect the 1 is taking the place of a minus sign. The extreme left-hand bit of any memory register or of any binary number is called the ‘Most Significant Bit’ or ‘MSB’ for short. If the MSB is 1 then you may be dealing with a negative number.

The Most Significant Bit can be found on the left hand of the 8 bit register.
Unfortunately negative numbers in an eight-bit register are not that simple. If 00001110 is 14 in decimal, you might expect that 10001110 would be -14 in decimal.

We can show that this does not lead to a satisfactory result. If you add:

\[
\begin{align*}
+ & \quad 14 \\
- & \quad 14 \\
\hline 
& \quad 00
\end{align*}
\]

the result is obviously zero.

Now add:

\[
\begin{align*}
00001110 \\
10001110 \\
\hline 
10010000
\end{align*}
\]

Quite obviously the result is not zero and so is unsatisfactory. A slightly more sophisticated method is needed.

Surprisingly we look to the car millennium for the answer. You have just bought a brand new Sierra/Maestro/Datsun (delete as applicable). The car is so new that the millennium actually reads 000000 on the clock. (I know this is unlikely even in a new car.) You are a little eccentric. You decide to reverse
out of the show-room. In fact you reverse for a whole mile!
What will you see on your milemeter?
You will, of course, see 999999. (Don’t say that you never
reversed a new car one mile to prove this!) You could say that
this reversing of one mile is the same as going -1 mile.
The same is true for binary in a memory register. If you take
one away from a line of zeroes (00000000), you will get - not of
course 99999999 - but its binary equivalent, 11111111. So:

11111111 = -1
11111110 = -2
11111101 = -3
11111100 = -4

and so on, just as if your car kept on reversing!
It is fairly easy to show that this works. Add:

\[
\begin{array}{c}
00000001 \\
11111111 \\
\hline
10000000
\end{array}
\]

(+1 decimal)
(-1 decimal)

1 00000000 (0 decimal)

The bit marked with the arrow is the ninth bit and will not fit
into an eight-bit register. So the result in the eight-bit register is
00000000, which is what you want.
What happens to the ninth bit?
There are times that you will want to keep this bit. It is called
the ‘carry’ bit. The carry flag in the P register will always show if
there was a suspected carry from an eight-bit register.
Sometimes, as we shall show, you may need to add a carry bit as
part of your sixteen-bit addition!
This method of representing negative numbers is called
‘two’s complement’. If you are ever worried why it works, think
of the car milemeter.
You will need a quick method of working out a negative
number in two’s complement. There is no real need to under-
stand why it works.
To find the negative number in two’s complement of +45
decimal:

+45 = 00101101 in binary

Now reverse the digits, i.e. change 1s to 0s and 0s to 1s:
\[ +45 = 00101101 \quad \text{(before)} \]
\[ 11010010 \quad \text{(after changing)} \]

\[ +45 = \]
\[ \quad 00101101 \]
\[ \quad 11010010 \]
\[ \quad +1 \]
\[ \quad \text{--------} \]
\[ \quad 11010011 = -45 \]

11010011 is -45 in the two's complement method of showing negative numbers. Just to prove to yourself that it works, add:

\[ 00101101 \quad (+45) \]
\[ 11010011 \quad (-45) \]

\[ \quad \text{--------} \]
\[ 1 \quad 00000000 \]

↑ carry

So the result in the eight-bit register is 00000000.

This 'short cut' to working out two’s complement also works if we have a negative number and want to know the equivalent positive number.

Remember the number &FA used in our branch instruction! Let us discover what this is in two’s complement.

The hexadecimal number &FA translates into binary as follows:

\[ \begin{align*}
1111 & \quad 1010 \\
F & \quad A
\end{align*} \]

&FA = 11111010

Now reverse the numbers:

\[ \begin{align*}
11111010 \\
00000101
\end{align*} \]

and add one:
This is +6, therefore 11111010 is -6.

The branch instruction in our program has told the program counter to go back six memory locations, to find where to branch (jump) to. Is this correct?

Look back at the last program.

The machine code generated from the beginning of the instruction ‘STA Screen_start,X’ to the end of the instruction ‘BNE More’ is:

9D 00 7C CA D0 FA

This is six bytes. Six memory locations have been used! So the number &FA after the machine code D0 for BNE is correct!

It is important that you understand this section of the book. If you have any doubts about why all this works - work through it again. It is quite satisfying that it ‘all comes out in the wash’.
CHAPTER 7

INTRODUCING NEW INSTRUCTIONS AND THE BBC OPERATING SYSTEM

The following two programs introduce a couple of new instructions and how to use the BBC Operating System.

```
10 REM ************************************************************************
20 REM * Alphabet Program *
30 REM ************************************************************************
40 MODE 7
50 LET Start__of__screen% = HIMEM
60 LET Line__of__screen% = 40
70 LET Total__of__alphabet% = 26
80 LET First__letter% = ASC("A")
90 HIMEM = HIMEM - 100
100 LET Start__of__program% = HIMEM
110 LET Start__of__display% = Start__of__screen% +
    (10 * Line__of__screen%)
120 REM ------------------------
130 LET P% = Start__of__program%
140 [ OPT 3
150 .Prog LDY #First__letter%
160 LDX #Total__of__alphabet%
170 .More TYA
180 STA Start__of__display%,X
190 INY
200 DEX
210 BNE More
220 RTS
230 ]
240 REM ------------------------
250 INPUT "Press Return "A$
260 CALL Start__of__program%
270 END
```

This program uses the ‘moving HIMEM’ method of reserving space for the machine code. As we need to know the start of the
screen memory location, we must store the value of HIMEM (line 50) before we move HIMEM. In MODE 7 the screen is forty characters wide, so the line counter (line 60) must be set to 40. We can now arrange for our display to appear below the assembler listing, unlike the previous program.

Two new instructions are introduced here: INY and TYA. ‘INY’ means ‘INcrement the Y register’. One is added to whatever is currently in the Y register. The 6502 chip has a very small number of possible machine code instructions, and no ‘increment the accumulator’ is provided. This is different from many other eight-bit microprocessors and a little inconvenient because, as we discovered earlier, most of the input/output from a computer goes through the accumulator, and there are several times when a quick increment of the accumulator is called for. So we must take a copy of the Y register and transfer it to the accumulator before outputting it. The Y register will be used for the incrementing as required.

‘TYA’ means ‘Transfer Y to Accumulator’. Whatever was in the Y register remains in the Y register - just a copy is transferred.

When you run this program you should see something like:

<table>
<thead>
<tr>
<th>Location</th>
<th>Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7B8C</td>
<td>OPT 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7B8C</td>
<td>A0 41</td>
<td>.Prog</td>
<td>LDY #First_letter%</td>
</tr>
<tr>
<td>7B8E</td>
<td>A2 1A</td>
<td></td>
<td>LDX #Total_of_alphabet%</td>
</tr>
<tr>
<td>7B90</td>
<td>98</td>
<td>.More</td>
<td>TYA</td>
</tr>
<tr>
<td>7B91</td>
<td>91 90</td>
<td>7D</td>
<td>STA Start_of_display% ,X</td>
</tr>
<tr>
<td>7B94</td>
<td>C8</td>
<td></td>
<td>INY</td>
</tr>
<tr>
<td>7B95</td>
<td>CA</td>
<td></td>
<td>DEX</td>
</tr>
<tr>
<td>7B96</td>
<td>D0 F8</td>
<td></td>
<td>BNE More</td>
</tr>
<tr>
<td>7B98</td>
<td>60</td>
<td></td>
<td>RTS</td>
</tr>
</tbody>
</table>

The ‘transfer’ instructions also include:

TXA   Transfer a copy of the X register to the Accumulator
TAX   Transfer a copy of the Accumulator to the X register
TAY   Transfer a copy of the Accumulator to the Y register

The increment instructions also include:
INX   Add one to the X register
INC   Add one to any specified memory address.
The INC instruction needs a knowledge of ‘memory addressing techniques’ which are handled a little later on (see page 000).

When you press the Return key, you will see the alphabet printed from Z to A on the tenth line of the screen. We had better investigate why!

As before the program is split into three sections: lines 10-110, lines 120-240, lines 250-270.

**Line 40** The MODE 7 is important as it ‘re-initialises’ the screen memory locations.

**Line 50** Unless moved, HIMEM, points to the start of screen memory.

**Line 60** This is used in counting the lines down.

**Line 70** This is the value for the counter which is going to be kept in the X register.

**Line 80** The code for the letter A will be stored in the Y register.

**Line 90** HIMEM can now be moved to allow room for the machine code. Note that ‘LET HIMEM = HIMEM - 100’ is *not* valid BBC BASIC. This is rather strange: the use of the LET here generates a syntax error, so avoid it!

**Line 100** Self-evident.

**Line 110** Decides where screen display will start.

**Line 130** Sets the program counter.

**Line 150** Puts A into the Y register.

**Line 160** As before, the X register is used to index the final screen position that the letter will appear in. The start of the tenth line is the base address for the index. The first letter that will appear will be the A. This will be twenty six screen locations from the left-hand margin.

**Line 170** The transfer that we need.

**Line 180** Stores a copy of the accumulator at the base address + the index.
Line 190 One is added to the Y register which at first holds the character code for A. After this instruction it will hold the character code for B which is &42 (66 decimal). It will then be ready for this character to be output to the screen.

Line 200 The index in register X goes down by one, which means that the letter B will appear immediately to the left of the A - and so on.

Line 210 The branch instruction.

Line 220 Return from subroutine.

Lines 250-270 Self-evident

If an assembler freak is looking over your shoulder, he will tell you that there are STX and STY instructions. Though these are similar to the STA instruction, working on the X and Y registers respectively, to use them properly needs a better understanding of ‘memory addressing techniques’. Also I wish to lead easily into the use of the BBC operating system - which requires input/output via the accumulator.

**BBC OPERATING SYSTEM CALLS**

Any manufacturer of any microcomputer must write an operating system which will allow humans to communicate with the microcomputer. This operating system will be in machine code. It may have been written in the assembly language for the microprocessor driving the computer. Sometimes it will have been written in a high level language and ‘compiled’ to machine code. Most manufacturers will give the user of the microcomputer access to some of the machine code routines that make up the operating system.

In the examples above, we have been directly accessing the screen, using our knowledge of the memory map of the screen in MODE 7. Obviously Acorn, who developed the BBC Microcomputer, have written machine code routines to do precisely this; otherwise you would not be able to access the screen at all in MODE 7. We need to know where in memory
Acorn have 'hidden' their machine code routines to access the screen in MODE 7.

Fortunately we do not have to play detective. The User Guide to the BBC Microcomputer on tells us on Page 452 exactly how to find many of the machine code routines that we want. It does not tell us the actual memory locations where the machine code that we want is. However we do not need this information. We need to know:
1. the 'entry points' to the BBC operating system
2. any registers of the 6502 chip that must be loaded with particular values before we use the entry point.

The entry points on the BBC are a series of memory addresses from &FFCE to &FFF7. You just treat these memory addresses as if they were the addresses of subroutines. You use the assembler instruction 'JSR ???', where 'JSR' means 'Jump to SubRoutine' and '????' is a four hexadecimal digit address or a label. (A subroutine is a piece of program that stands on its own, but which can be incorporated as part of any bigger program.)

When you use the BBC operating system machine code, even though you will be incorporating the subroutines into your own programs, you will not discover just where in memory the whole of each subroutine lies. Nor will the actual machine code appear as part of your own machine code. This is not important.

We will be concerned with:

<table>
<thead>
<tr>
<th>Name of routine</th>
<th>What it does</th>
<th>Address to jump to</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSRDCH</td>
<td>Sees which character has been used at the keyboard</td>
<td>FFE0</td>
</tr>
<tr>
<td>OSASCI</td>
<td>Puts a character on the screen from the accumulator</td>
<td>FFE3</td>
</tr>
<tr>
<td>OSNEWL</td>
<td>Sends a new blank line to the screen</td>
<td>FFE7</td>
</tr>
<tr>
<td>OSWRCH</td>
<td>The machine code version of the VDU command in BASIC</td>
<td>FFE0</td>
</tr>
</tbody>
</table>
OSWORD  The machine code version of some of the facilities of the BBC Micro

OSBYTE  The machine code entry to the 'FX' calls

Now study this program:

```
10 REM ******************************
20 REM * Use of Operating System  *
30 REM ******************************
40 MODE 7
50 LET Start_of_program% = &3000
60 LET First_letter = ASC("A")
70 LET BBC_operating_system = &FFE3
80 LET Total_of_alphabet = 26
90 REM ------------------------
100 LET P% = Start_of_program%
110 [ OPT 3
120 .Prog LDY #First_letter
130     LDX #Total_of_alphabet
140 .More TYA
150     JSR BBC_operating_system
160     INY
170     DEX
180     BNE More
190     RTS
200 ]
210 REM ------------------------
220 INPUT "Press return "A$
230 CALL Start_of_program
240 END
```

I have chosen to use the BBC operating system routine at &FFE3 called OSASCI. This is the routine that writes to the screen without any of the frills of the BASIC VDU command. So in the first section of the program I have declared the address of this routine. Generally you will find in books and magazine articles that authors will refer directly to OSASCI, OSWRCH, etc. When they refer directly to these routines, they mean the memory location where these routines can be 'called' into life.
Though the first third of the program differs a fair degree from the previous program, the only difference in the machine code is that 'STA Start_of_display%,X' becomes 'JSR BBC__operating__system'. This is because in this program we are leaving it to the BBC operating system to get the characters actually out to the screen. The OSASCII routine needs the character code of the character that you want on the screen to be placed into the accumulator ready for output.

When you look at the assembly listing you will notice that it even uses exactly the same number of bytes of memory. Only when you press Return will you notice the difference: the alphabet appears from A to Z starting at the left-hand margin.

Only twenty six characters appear because the X register is being used as a counter. The X register is not being used as an index here. The OSASCII outputs characters from the position of the cursor. After you have produced the assembly listing the cursor is returned to the left-hand margin, and so the first character output, the A, appears there. The BBC operating system automatically moves the cursor one position to the right after the call to the OSASCII routine. It is therefore correctly positioned for when the letter B is to be output. As before, the Y register is being incremented, so that each time it holds the ASCII character code value of the next letter.

If the lengths of the two programs for outputting the alphabet are the same, is there any real difference between them? Apart from producing a slightly different display, unfortunately there is. The program using the BBC operating system will take considerably longer to execute than the direct use of the screen memory map. This is because you have to include all the operating system machine code involved in the OSASCII routine. Unfortunately Acorn do not tell you the number or length of the instructions they have used to make OSASCII work, nor the time they take, and so it is very difficult to assess properly the difference in speeds between the two routines. It would be reasonable to assume that the direct addressing of the screen was at least twice as fast as using the OSASCII routine.

However, this is not generally significant unless you are going to write very fast games or use your computer to communicate with outside devices and machinery. But a word of warning: Acorn have made it very clear that they will only guarantee that the machine code you write will work on all future upgrades of the BBC Microcomputer if you use their machine code routines as appropriate. This is particularly true of screen addressing from the 'other side' of the 'tube'.

78
However, for those of us with standard Model As and Model Bs, direct use of the screen memory maps is possible.

The OSASCI routine happily works in any mode, not just MODE 7. In fact, because in other modes the screen memory maps are so odd, you would do well to always use OSASCI or one of the other BBC operating system calls we shall be introducing later.

**SUMMARY**

The assembler mnemonics that you should now understand include:

- **JSR** Jump to Sub-Routine
- **RTS** ReTurn from Sub-routine
- **INY** INcrement the Y register
- **INX** INcrement the X register
- **DEY** DEcrement the Y register
- **DEX** DEcrement the X register
- **TYA** Transfer (copy) the Y register to the Accumulator
- **TXA** Transfer (copy) the X register to the Accumulator
- **TAY** Transfer (copy) the Accumulator to the Y register
- **TAX** Transfer (copy) the Accumulator to the X register
- **LDA #??** LoaD the Accumulator with an actual number
- **LDX #??** LoaD the X register with an actual number
- **LDY #??** LoaD the Y register with an actual number
- **STA ???.X** STore (copy) the Accumulator into a memory address indexed by the X register
- **BNE** Branch (jump) if the zero flag has Not been set.
CHAPTER 8
MORE ABOUT THE 6502 PROCESSOR

In many ways the 6502 is a curious beast. It has, as part of its design, a method of looking at chunks of memory as if they were pages of a book.

This idea of 'pages' should not be totally unknown to BBC Micro owners. The BBC Micro lets you page ROMs, so that they lie beside each other with the same memory addresses. For instance the wordprocessor chips 'View' or 'Wordwise' lie side by side with the BASIC chip using the same part of memory and memory addresses. They look, to the 6502, like pages of a book, though each has the same page numbers, as if the 'story' had alternative versions.

However, this is not the same as the paging used by the 6502 chip in its general view of memory. Memory is seen as blocks of 256 bytes. So the first page is from memory location 0 to 255. However, to confuse matters, the first page of memory locations is called the zero page or page zero. It is as if your book started at Page 0, not Page 1, and each page had exactly the same number of words on it.
Generally the page organisation of memory should not worry you. However it can affect the time taken by the processor to perform an instruction. Consider the BNE instruction that we are now familiar with. If the branch crosses a page boundary, in other words involves turning over a new page, then it can nearly double the time taken to perform the branch. In most of your machine code programming timing may not be significant - but there are times when an instruction being slowed down can have unforeseen affects.

There is also special significance given by the 6502 chip to page zero. This is a powerful feature of the chip.

**ZERO PAGE ADDRESSING**

Generally on an eight-bit microprocessor, if you wish to specify a memory address, you will need four hexadecimal digits to do it. This is because the eight-bit microprocessors allow two of their eight-bit registers to hold the sixteen bits needed to specify all the possible memory locations from &0000 to &FFFF.

However, it can grow to be expensive on memory if, every time you need to specify an exact memory location, you find yourself using two precious eight-bit registers. So the 6502 allows you to ‘forget’ the high part of the memory address.

Let us consider the STA instruction. As you know, STA means STore the Accumulator. ‘STA &257F’ means take a copy of the accumulator and store it in memory location 257F hexadecimal.

If we forget the &25 (high) part of the memory address then our instruction now reads STA &7F.

The 6502 chip then assumes that you want memory location &007F. Of course, when the high part of the address is &00, then the address has to be in page zero. This is why it is called zero page addressing.

The important point is that the machine code for STA &257F is ‘9D 7F 25’ (three memory locations used). But STA &7F is ‘85 7F’ (two memory locations used).

Zero page addresses can be used with most instructions, except JMP and JSR which require the full sixteen-bit address whatever the circumstances!

You will have gathered that finding memory space for your machine code can be a problem in the BBC Micro. This means that using zero page addresses by forgetting the high part of the address may just save you the space to fit your program in. But there are only 256 possible zero page addresses from &0 to &FF (0 - 255). The BBC operating system uses most of the zero page addresses.
Thoughfully the designers have allowed you the use of addresses &70 to &8F. This allows us to amend our last program a little:

```
10 REM ****************************************
20 REM * Zero Page Addressing       *
30 REM ****************************************
40 MODE 7
50 LET Start_of_program = &3000
60 LET First_letter = ASC("A")
70 LET BBC_operating_system = &FFE3
80 LET Total_of_alphabet = 26
90 LET ?&70 = First_letter
100 LET ?&71 = Total_of_alphabet
110 REM ------------------------
120 P% = Start_of_program
130 [ OPT 3
140     LDY &70 /First_letter/ 
150     LDX &71 /Total_of_alphabet/ 
160 .MoreTYA
170     JSR BBC_operating_system
180     INY
190     DEX
200     BNE More
210     RTS
220 ]
230 REM ------------------------
240 INPUT "press return "A$
250 CALL Start_of_program
260 END
```

Notice that Lines 90 and 100 are different. In BASIC, we have made the contents of zero page memory locations &70 and &71 hold the first letter and the total of alphabet. Remember that the ‘?’ is used by BBC BASIC to mean ‘the contents of’.

Then we have accessed these zero page locations in lines 140 and 150. These lines also introduce you to the method of commenting assembly programs.

**COMMENTS**

Everything that you write between the backwards slash marks ‘/’ is treated as a comment. In MODE 7 the backwards slash
appears on the screen as a half. Don’t worry: it still sections off your comments.

In the program above the comments were essential, otherwise the use of the zero page addresses (&70 and &71) in the assembly program would seem to have little purpose.

The general rule about commenting your program is that YOU CAN NEVER HAVE TOO MANY COMMENTS.

You can never have too many comments!

When you return to a program after a few months you will often find that you have forgotten how or why you wrote some assembler mnemonics. It happened to one of the young lads who helped with this book. If, when you return to your program, you just see a mass of assembly language without any clues as to what it does or how it does it, then either you will have to ‘reinvent the wheel’ or put an ice pack on your head as you try to puzzle out your own brilliance.

The only constraint on commenting should be the memory space for the ‘source program’. (The source program is what we call your assembly program before the mnemonics have been converted into machine code. When we run your source program and do the conversion into machine code (producing the assembly program listing), then we produce the ‘object program’. The object program is another word for the machine code version of your program.)

The run of the program above is:
When you press return, the same alphabet appears. But now, without altering the assembly program in any way, you can 'fiddle' with what you see on the screen. For instance type into the BBC micro:

LET ?&70 = ASC("B")
LET ?&71 = 25
CALL Start_of_program

(If you have pressed the Break key, then you will need to type CALL &3000.)

The letters from B onwards appear. If you want most of the BBC character set to appear then type into the machine:

LET ?&70 = ASC(" ")
LET ?&71 = 95
CALL &3000

OK, this is silly. But it does show you some of the ways of using zero page addresses. In particular note that, provided that you do not 'overwrite' the memory addresses where your machine code is stored, then, unless you turn off the machine, your machine code program can always be reCALLed. You do not need to reassemble it. You can change the values that the program uses by changing the values (using BASIC) in the zero page memory locations.
CHAPTER 9

OTHER BBC OPERATING SYSTEM CALLS

OSASCI is the easiest operating system ‘call’ to use, but it only allows you to send ASCII characters to the screen. Most of the time you will want to send commands to the screen as part of the BASIC ‘VDU commands’: you will need OSWRCH.

OSWRCH

OSWRCH (jump address = &FFE) means Operating Sytem for WRiting CHaracters to the screen. For instance if you wish to define a text window, the VDU command is ‘VDU 28’ followed by four numbers indicating the bottom left X position, bottom left Y position, top right X position, top right Y position of the text window. The User Guide gives on Page 387 the example of:

VDU 28,5,20,30,12

To create this in machine code you would need to use the
OSWRCH routine five times to send the numbers 28, 5, 20, 30 and 12.

OSWRCH, like many routines, needs the character you want to send to be placed into the accumulator before outputting it. Somewhere your assembly program needs to look like this:

**Version 1**

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>LDA #28</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>JSR &amp;FFEE /OSWRCH routine used/</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>LDA #5/to define a text window/</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>JSR &amp;FFEE</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>LDA #20</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>JSR &amp;FFEE</td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>LDA #30</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>JSR &amp;FFEE</td>
<td></td>
</tr>
<tr>
<td>280</td>
<td>LDA #12</td>
<td></td>
</tr>
<tr>
<td>290</td>
<td>JSR &amp;FFEE</td>
<td></td>
</tr>
</tbody>
</table>

You do not really need to write the OSWRCH routine five times. You can write it once and jump back to it another four times, collecting the data you want from a zero page address indexed by the X register.

**Version 2**

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>LDY #5</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>LDX #0</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>.textw LDA &amp;70,X</td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>JSR &amp;FFEE /OSWRCH/</td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>INX</td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>DEY</td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>BNE textw</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

It is important to note that only the X register can be used to index a zero page memory location. The Y register is being used as a counter to control the number of times we loop back to the label 'textw'. We could get rid of this use of the Y register by storing the numbers for the VDU command not from &70 up
to &74, but down from &74 to &70. Then as we decremented the index register X we could also use it as a counter coming down from 5 to 0. The same BNE instruction would be used:

### Version 3

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>210</td>
<td>LDX #5</td>
</tr>
<tr>
<td>220</td>
<td>.textw DEX</td>
</tr>
<tr>
<td>230</td>
<td>LDA &amp;70,X</td>
</tr>
<tr>
<td>240</td>
<td>JSR &amp;FFEE /OSWRCH/</td>
</tr>
<tr>
<td>260</td>
<td>BNE textw</td>
</tr>
<tr>
<td>270</td>
<td>RTS</td>
</tr>
</tbody>
</table>

Notice that there has been a slight rearrangement of the order of the instructions. Though the DEX instruction will set the zero flag, we get away with not testing for the zero flag till line 260 because the JSR instruction does not affect the flags in the P register, and at no time are we loading the accumulator with zero. This short cut program therefore has a ‘bug’ and we cannot use it if part of our output stream of numbers via the accumulator contains a zero.

For Versions 2 and 3, you must remember to fill the zero page addresses with the right data using the BASIC ‘?’ command.

When run, Version 3 of the program produces the machine code as follows:

A2 05 CA B5 70 20 EE FF D0 F8 60

It uses only eleven memory locations. This is far better than the twenty five memory locations of the first version and the fourteen memory locations of the second.

The bug in Version 3 can be corrected if we introduce another kind of ‘JUMP’ instruction, the ‘BPL’ mnemonic. This allows you to ‘Branch if the accumulator is PLus’. It means ‘JUMP to another address if the accumulator has a positive number in it’. If you remember, we can tell the difference between a negative and positive binary number by looking at the ‘sign’ bit: a 0 in the extreme left-hand bit means a positive number, and a 1 means a negative number. Strangely, the binary number for zero, 00000000, will be considered as positive by this instruction because it has a 0 in the most left-hand bit! So Version 4 of the program (without the bug) can be written as:
Version 4

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>210</td>
<td>LDX #4</td>
</tr>
<tr>
<td>220</td>
<td>.textw LDA &amp;70,X</td>
</tr>
<tr>
<td>230</td>
<td>JSR &amp;FFEE /OSWRCH/</td>
</tr>
<tr>
<td>240</td>
<td>DEX</td>
</tr>
<tr>
<td>260</td>
<td>BPL textw</td>
</tr>
<tr>
<td>270</td>
<td>RTS</td>
</tr>
</tbody>
</table>

You will gather that generally the VDU command in BASIC is easier to use than a machine code routine to do the same thing. At the end of the book we have included some programs which do use the OSWRCH for machine code graphics examples.

**Warning:** A lot of the VDU commands require a series of numbers to be sent out, such as VDU 28, which requires another four numbers to be output. You will wreak havoc with your machine code if you forget to use the OSWRCH call on the whole of the output stream of numbers you need to send with your VDU command. In the example above, we might have sent out only the 28,5,20,30 of the text window and then decided we wanted to clear the screen with a VDU 12. Of course the 12 would be considered as the last part of the VDU 28 command, and not as a separate VDU 12 command - or its OSWRCH equivalent.

The situation becomes more confused when we use the OSWRCH to send out the VDU versions of the BASIC 'PLOT' commands. The PLOT values range from 0 to 1279 but the accumulator range of values is from 0 to 255.

To overcome this we need to split the graphics screen position into two numbers. We do this with the BASIC 'DIV' and 'MOD' commands. Let us consider the BASIC command: 'MOVE 350,427'. This is the same as 'PLOT 4,350,427,' and that is the same as 'VDU 25,4;350;427;'.

The 350 and 427 must be split down further. If you type into the computer:

```
PRINT 350 DIV 256
PRINT 350 MOD 256
```

you will see the answers 1 and 94. And if you type again:
PRINT 427 DIV 256
PRINT 427 MOD 256

you will see the answers 1 and 171.

We now have a new version of the VDU 25 command which means the same as the PLOT, MOVE or VDU 25 above:

VDU 25,4,94,1,171,1

Your OSWRCH call for the simple graphics MOVE 350,427 involves you in sending six numbers to OSWRCH via the accumulator: the 25,4,94,1,171 and 1.

**Why MOD and DIV?**

![Diagram](image)

**Using DIV & MOD**

As the accumulator can only handle the 256 different numbers from 0 to 255, we split the graphics numbers up into a whole number of 256s and the remainder. The whole number of 256s (made by the DIV) becomes the 'high' part of the number. The remainder (made by the MOD) becomes the 'low' part of the number:

\[
(1\times256) + 94 = 350
\]

\[
(1\times256) + 171 = 427
\]

As you remember from high and low parts of addresses, for a curious reason, the 6502 likes the low part to be sent out before the high part. Do not be confused by the fact that the high part of
the number - a 1 in this case - has a smaller number in it than the
low part of the number - 94 in this case. Remember the 1 means
1*256 which actually makes it the higher part of the number.

A final word of warning: if you want to PLOT a number like
53,427, then you must still do the DIV and MOD on the number
53. The OSWRCH numbers for:

MOVE 53,427

are 26,4,53,0,171,1.

The zero marked here means that there are no 256s in the high
part of the number. The computer needs this fact to make the
OSWRCH version of VDU 25 work properly.

**OSRDCCH**

OSRDCCH (jump address = &FFE0) means Operating System for
ReaDing keyboard CHaracters. By comparison it is very easy to
use. A call to the routine will result in the ASCII code for the
letter pressed on the keyboard being found in the accumulator.
If you use OSASCI and OSRDCCH together you can type out the
keyboard.

```
100 .Again JSR &FFE0 /OSRDCCH/
110      JSR &FFE3 /OSASCI/
120      JMP Again
```

This is not very clever. A good use of the OSRDCCH call is in
games. You often want to get the machine to respond as fast as
possible to the depression of a key. Though BBC BASIC is very
fast, it may not be fast enough.

There are some important conditions that you must take into
account when you use OSRDCCH. If there has been a successful
reading of the keyboard then the carry flag will be zero (not
set). If either there has been an error generated in reading the
keyboard or the Escape key has been pressed, then the carry
flag will have a 1 in it, and the error number will be in the
accumulator.

The USER GUIDE (Page 456) tells you that you must
acknowledge the Escape key by doing a special OSBYTE call.
In this OSBYTE call the accumulator must be set to &7E - more of
that later. The ASCII value for the Escape key is &1B (27 decimal).
If you are looking for a particular key being pressed, then the assembler mnemonic CMP is useful. It means 'CoMPare'. You can compare the number or character code in the accumulator with any number or character code. If the codes are the same, then the zero flag is set (has a 1 in it). So:

CMP #&41

could be used to spot the letter A, as &41 (65 decimal) is the ASCII code for A. Notice that the '#' sign is used to show that the &41 is an 'immediate' number for direct comparison and not a zero page address. The little program above could now be amended to:

```
70 LET Start_of_program = &3000
80 P% = Start_of_program
90 [ OPT 3
100 Again JSR &FFE0 /OSRDCH/
110 CMP #&41 /the letter "A"
120 BNE Again
130 JSR &FFE3 /OSASCI/
140 JSR &FFE7 /OSNEWL/
150 RTS
160 ]
170 CALL Start_of_program
180 PRINT "At last you have found it"
190 END
```

Anything else, except pressing the Escape key, will be ignored. Finally you find the letter A, and the message appears!

**OSBYTE**

The OSBYTE (jump address = &FFF4) routine is the machine code version of the *FX calls. For those, such as myself, who have the early version of the BASIC chip, it provides the only way in which you can pass a variable into an FX call. This is because in the early BASIC the following is forbidden:

INPUT "Printer type" Code
*FX 5, Code
This can be a little irritating for those used to BASIC, as otherwise a series of IF ... THEN statements are needed. The following program may help:

```
10 REM *****************************************************
20 REM * OSBYTE routine for printer    *
30 REM *****************************************************
40 MODE 7
50 LET Start_of_program = &D00
60 LET P% = Start_of_program
70 [ OPT 3
80     CLD
90     LDA #5
100    LDX &70
110    JSR &FF44   /OSBYTE/
120    RTS
130 ]
140 REPEAT
150 INPUT"Printer type (1 or 2) "Type
160 UNTIL Type = 1 OR Type = 2
170 LET ?&70 = Type
180 *FX15: REM flush buffers
190 CALL Start_of_program
200 VDU 2
210 PRINT "IT WORKS"
220 VDU 3
230 END
```

Three important points emerge here:
1. Put your machine code at the start of a program and assemble it first. This will avoid, among other things, problems with the program counter later.
2. When using the OSBYTE call, you should use the CLD instruction as well. The CLD instruction means ‘CLear Decimal flag’. The 6502 chip can operate either in ‘decimal mode’ or ‘binary mode’. Everything we have done so far has been in binary mode, but it is possible to do maths with the 6502 chip in decimal mode. The OSBYTE may not work if the computer is in decimal mode. The decimal flag in the P register shows this. So to make sure, we CLD - CLear the Decimal flag and reset back to binary mode.
3. With an OSBYTE call, you will be using the accumulator, X
and Y registers. If the *FX call only has one number following it, then that number goes into the accumulator before the OSBYTE is called. This would be true of *FX18. If there are two numbers in the *FX call separated by a comma, then the first number goes in the accumulator and the second into the X register. Then call the OSBYTE routine. The printer routine above with *FX5,1 is an example. When an OSBYTE has three numbers after it, such as would be the case with *FX139,1,0, then as before the first number goes into the accumulator, the second into the X register and the third into the Y register. Then the OSBYTE is called.

OSBYTES can be very powerful.

There are some very powerful and useful OSBYTE routines such as reading the character at the text cursor position (OSBYTE 135) or putting extra characters into the keyboard buffer (OSBYTE 138). I suggest that you read Pages 429-436 of the USER GUIDE in more detail, now that you understand the OSBYTE a little better. You will be pleasantly surprised.

OSWORD

There are several features of the BBC Micro which are very impressive, such as the sound chip. But programming it in machine code via the operating system is not easy because the envelope needs fourteen different ‘parameters’ passed into it. A parameter is just a value of a variable that the particular
feature needs. The fourteen parameters cannot easily be passed via the accumulator, X and Y registers. So a different method is used. The X and Y registers will be filled by you with the base address of a table of memory locations which will hold all the data needed by, for instance, the envelope command. This table of memory locations is called a 'parameter block'.

**Using a parameter block**

To define an envelope needs fourteen parameters. The OSWORD (jump address &FFF1) call number for defining an envelope is OSWORD 8. The '8' is placed into the accumulator, and then you must find fourteen consecutive free memory locations for your fourteen variables in the block of memory locations called a parameter block (see Figure 10). The Y register will hold the high part of the memory address and the X register the low part.

---

**Fig 10 USE OF OSWORD PARAMETER BLOCKS**
This is an excellent opportunity to use the DIM statement again. Of course you will need to use the DIV and MOD commands again. Look at this example:

```
10 REM ****************************
20 REM * OSWORD USE FOR ENVELOPE *
30 REM ****************************
40 MODE 7
50 DIM Start_of_program% 25
60 DIM Parameter% 14
70 LET P% = Start_of_program%
80 LET Low% = Parameter% MOD 256
90 LET High% = Parameter% DIV 256
100 [ 
110 OPT 3
120 LDA #8
130 LDX #Low%
140 LDY #High%
150 JSR &FFF1 /OSWORD/
160 RTS
170 ]
180 FOR C = 0 TO 13
190 READ Data
200 LET Parameter%?C = Data
210 NEXT
220 CALL Start_of_program%
230 SOUND 1,1,100,230
240 DATA1,2,4,-3,4,10,20,10,127,0,0,-3,126,125
250 END
```

The sound generated is the familiar siren sound.

Notice that I have used a BASIC indirection operator, '***?', for putting the values for the envelope into the parameter block. This program also mixes BASIC and machine code, when the particular task would have best been done totally in BASIC: but it illustrates the use of a parameter block.

The sound command, when used via the OSWORD 7 call, also has a parameter block. But because some of the four sound parameters can take values greater than 255, all the four sound parameters are divided up into high and low bytes. Look at the USER GUIDE, Page 461, for exact details.

Once you have mastered the idea of the parameter block, you will find the OSWORD calls as much fun as the OSBYTE calls!
The OSWORDS perform a number of miscellaneous operations.
CHAPTER 10

ASSEMBLER MATHS

Till now in the book I have resisted any ‘hard’ maths. It is possible to survive without much knowledge of maths, particularly in the BBC Micro. You can retreat to BASIC every time you need to do some maths and use the facilities of the assembler to mix the BASIC into your program. But there are a series of assembler mnemonics which refer directly to maths.

ADDITION

Refer back to the early section on binary addition (page 3), if in doubt. The 6502, unlike other microprocessors such as the Z80 chip, can perform only one kind of binary addition: `ADdition with Carry` (ADC).

This means that it will always add two numbers together and what is in the carry flag of the P register. There is no 6502 instruction which does addition without trying to add in the carry flag as well. Whether the carry flag has a 1 or a 0 in it will depend on several factors as there are thirteen assembler instructions which ‘influence’ the carry flag.

Obviously there are several occasions when you want to add two numbers together without adding some strange ‘carry’ from nowhere. The 6502 instruction CLC means CLean the Carry flag. This has the effect of putting a 0 into the carry flag. Obviously, then, the instruction ADC can be used safely because we have nothing being carried.

So before you do assembler addition, always clear the carry flag with CLC.

Now the following program will produce the correct result:

```
10 REM ************************************************
20 REM * Simple assembler addition *  
30 REM ************************************************
40 MODE 7
50 DIM Start__of__program% 15
```
60 LET First_number% = 30
70 LET Second_number% = 45
80 LET P% = Start_of_program%
90 REM ------------------------
100 [ 110 OPT 3
120 LDA #First_number%
130 CLC
140 ADC #Second_number%
150 STA &70 /*zero page address*/
160 RTS
170 ]
180 CALL Start_of_program%
190 PRINT; "The answer is ");&70
200 END

The binary addition that has been done is:

00011110 (30)
+00101101 (45)
+ 0 (carry flag reset to zero)

-----------
01001011 (75)

The machine code that has been generated is:

A9 1E LDA #First_number%
18 CLC
69 2D ADC #Second_number%
85 70 STA &70
60 RTS

The answer of '75' is of course displayed.

But now let us look more closely at this program. Firstly we can only add together numbers that can be stored in the eight bits of a computer register. If we had tried to load the accumulator with too large a number we would have caused an assembler error.

But we cannot even use all the eight bits of a computer register because the extreme left-hand bit (the MSB) is also used as the 'sign' bit. As you will remember from 'two's
complement’, if the MSB is 0 we have a positive number and if the MSB is 1 we have a negative number. So 00011110 meant that we had a positive number of value 30. This is because:

\[ 00011110 = (+30) \]

↑

Sign bit (positive)

As we are left with use of just seven bits for our positive numbers, then the total range of positive numbers we can add in our eight-bit register is from 00000000 (0) to 01111111 (127).

Similarly the full range of negative numbers in our eight-bit register is from 11111111 (-1) to 10000000 (-128).

Now let us look at a couple of problems. Firstly try making the First_number% 300. You will get a ‘byte’ error telling you that the particular assembler mnemonic you are using (LDA #???) will only work with a number in an eight-bit byte - and 300 needs at least nine bits to store it.

Now make the First_number% 120 and the Second_number% 100. Both 100 and 120 can be stored within seven bits. The result printed out will be 220. This is a number that needs at least eight bits if you are not using the sign bit properly. Do not be misled by this. The ‘?’ command in Line 190 does not spot use of the MSB to show the sign of the number. It simply records the decimal content of the register, ignoring the fact that you may be doing assembler maths with a sign bit.

So, if we are dealing with a sign bit, though the result looks correct, an error has in fact crept in. You of course have to decide whether you are doing ‘signed’ arithmetic or not - but if you are, then you had better read on. If we add

\[
\begin{align*}
01111010 & \quad (120) \\
+ \quad 01100010 & \quad (100) \\
\hline
11011100 & \quad (In \ signed \ computer \ addition \ the \ answer \ is \ -36, \ not \ 220) \\
\uparrow & \quad (-36) \\
\text{Sign bit (negative)}
\end{align*}
\]

We describe this situation as ‘overflow’. The eight-bit register is not large enough to hold the full result and the sign bit. The
effect of overflow here has been to turn the sign bit from positive to negative, even though we are adding two positive numbers. We can explore this overflow in our program:

```
10 REM **********************************************
20 REM * Simple assembler addition    *
30 REM **********************************************
40 MODE 7
50 DIM Start_of_program% 15
60 LET First_number%   = 100
70 LET Second_number%  = 120
80 REM -----------------------
90 FOR Pass = 0 TO 3 STEP 3
100 LET P% = Start_of_program%
110 [   OPT Pass
120 .prog LDA #First_number%
130    CLC
140    ADC #Second_number%
150    BVS Text
160    RTS
170 .Text LDA #ASC(“V“)
180    JSR &FFE3 /OSASCII/
190    RTS
200 ]
210 NEXT
220 REM -----------------------
230 INPUT “Press return “ A$ 
240    CALL prog
250    END
```

Notice that Lines 150 to 180 are now different.

**Line 150** This means ‘branch’ if overflow flag is set to 1. The computer can detect the overflow condition in computer maths, and will put a 1 in the overflow flag of the P register if a possible overflow has occurred. This branch uses that fact.

**Line 160** The branch is to the label ‘Text’. If the overflow flag is not set to 1, then the program will continue directly to the RTS instruction here.

**Line 170** I am putting the letter V into the accumulator in a short
cut method which mixes BASIC and assembler. If the branch to the label Text succeeds, then the letter V will appear on the screen.

**Lines 180-190** This outputs the letter V via the OSASCI and returns to BASIC.

This program is our first working example of a forward reference to a label. For this reason the assembler needs to pass through the source code twice. I have made two passes with the FOR ... NEXT loop. On the first time in the loop the variable pass will equal zero. Because I use STEP 3, the second time through the loop will be the last. The variable pass will now be 3.

I suggest you use the variable pass to control what type of listing you get with the OPT command. On the first pass you do not want a listing. Nor do you want possible label errors reported because you know that forward reference labels will not be found on the first pass. So on the first pass we want OPT 0, but on the second pass we want a full error listing, so we need OPT 3. The FOR ... NEXT above does this.

Notice that the program counter P% must be placed inside the loop. You will produce some ‘interesting’ results if you forget this.

Notice also that there is a label ‘prog’ at the beginning of the program which will provide the CALL address.

When you run the program you will produce the machine code:

A9 64 18 69 78 70 01 60 A9 4F 20 E3 FF 60

When you press Return, the letter V will appear showing that the overflow flag has been set. Now change the values of the two numbers back to 30 and 45. You will not get the V when you run the program because there is no overflow. Experiment with some other numbers such as -100 and -28. With the negative numbers you will get a byte error. This has to do with the number of bytes the BBC Micro uses to handle all numbers. Don't worry!

**Warning:** There is a label error that can occur in the BBC assembler. It will not spot the same label being used twice in the label field. This should cause a scream of outrage from the assembler. Instead it just takes the latest memory location value
of the label as the one you want. Assemble the above program but make Line 190 'Text RTS'. Havoc will result. If you look carefully, you will notice that the machine code for 'BVS Text' is now 70 06. Count forward six bytes, and you will see this is the RTS instruction at the end. The Text label you wanted has been missed out, and the ambiguous use of the label has not been spotted.

The opposite of BVS is BVC. BVC will cause a branch if the overflow flag has not been set to 1, but is 0. It means Branch on overflow flag being Cleared to zero.

Overflow can occur when we are dealing with negative numbers in two's complement. Consider, in two's complement, the following little sum:

\[
\begin{align*}
10000111 & \quad (-121) \\
+ 10010000 & \quad (-112) \\
\hline
1 00010111 & \quad (seems to be +23)
\end{align*}
\]

↑
Positive sign bit
Ninth bit is lost

Because the visible result in the eight-bit register is 00010111, it seems as if adding the two negative numbers has resulted in a positive number. Overflow has struck again.

All is not lost. We can always use two memory locations one after the other to provide sixteen possible bits for storing our numbers. In that we can use slightly larger numbers. The MSB (sign bit) is now the extreme left-hand bit of all the sixteen bits. Add 350 and 450:

\[
\begin{align*}
00000001 & \quad 01011110 \quad (350) \\
+ 00000001 & \quad 11000010 \quad (450) \\
\hline
00000011 & \quad 00100000 \quad (800)
\end{align*}
\]

↑ Sign bit (positive)

We need to look at this sum as two separate sums in the eight-bit registers. Let us tackle the low part of the number first. First we clear the carry flag with the CLC instruction. The sum now is:
01011110
+ 11000010
+ 0 (no carry)

-----------
1 00100000

↑ This ninth bit is carried

When adding the two low halves of the numbers together we discover that we need a ninth bit. In fact this ninth bit is put into the carry flag of the P register. So the result of this sum is that the carry flag is set to 1.

We now do the sum for the high halves of the numbers:

00000001
+ 00000001
+ 1 (keeping the carry)

-----------
00000011

If we now put together the results from the two halves we get:

00000011 00100000
  high   low

This is exactly the same as the sixteen-bit addition we did above. The program below will do the addition:

10 REM ********************
20 REM * bigger addition with carry  *
30 REM ********************
40 MODE 7
50 LET Number1 = 350
60 LET Number2 = 450
70 LET High_number1 = 350 DIV 256
80 LET Low_number1 = 350 MOD 256
90 LET High_number2 = 450 DIV 256
100 LET Low_number2 = 450 MOD 256
110 LET P% = &3000
120 REM ------------------------
125 [ OPT 3
130 .addup LDA #Low_number1
<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>140</td>
<td>CLC</td>
</tr>
<tr>
<td>150</td>
<td>ADC #Low_number2</td>
</tr>
<tr>
<td>160</td>
<td>STA &amp;70 /low result is stored/</td>
</tr>
<tr>
<td>170</td>
<td>LDA #High_number1</td>
</tr>
<tr>
<td>180</td>
<td>ADC #High_number2</td>
</tr>
<tr>
<td>190</td>
<td>STA &amp;71 /high result is stored/</td>
</tr>
<tr>
<td>200</td>
<td>RTS</td>
</tr>
<tr>
<td>210</td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>INPUT “Press return “A$</td>
</tr>
<tr>
<td>230</td>
<td>CALL addup</td>
</tr>
<tr>
<td>240</td>
<td>LET ?&amp;72 = 0</td>
</tr>
<tr>
<td>250</td>
<td>LET ?&amp;73 = 0</td>
</tr>
<tr>
<td>260</td>
<td>PRINT ;“Answer is “ !?&amp;70</td>
</tr>
<tr>
<td>270</td>
<td>END</td>
</tr>
</tbody>
</table>

Note here the use of the ‘Pling’ operator - the ‘!’ sign in Line 260.

BBC BASIC takes four bytes of memory location to store an integer number. The ! can be used to access directly those four bytes starting at the memory address given. In plain English Line 260 reads, ‘PRINT “The answer is " the number in the contents of the 4-byte memory location starting at &70’.

Remember that the ‘?’ means ‘the contents of’.

To make the 4-byte memory location ‘safe’, I have put zeros into &72 and &73. The four bytes now are:

- &70 low part of result 00100000
- &71 higher part of result 00000011
- &72 next higher set to zero 00000000
- &73 next higher set to zero 00000000

The full 4-byte number is:

0000000000000000000000001100100000

† Sign bit

The answer is +800.

It is possible that we might wish to add two sixteen-bit numbers which themselves would generate a carry. We can either store this carry as part of the result or detect the overflow.
condition (if we were trying to store the result in sixteen bits).

**SUBTRACTION**

The problems arising here are very similar to those in addition. But you must also be familiar with the two’s complement method of representing negative numbers.

A simple way of looking at it is that all instructions are exactly the reverse of addition. So instead of clearing the carry flag before addition, we set the carry flag to 1 before subtraction.

Subtraction is the reverse of addition.

The instruction to do this is ‘SEC’. Otherwise the pattern of assembler mnemonics is much the same. The program above needs little alteration to turn it into a subtraction program.

```
10 REM ****************************************************
20 REM *Simple assembler subtraction    *
30 REM ****************************************************
40 MODE 7
50 DIM Start_of_program% 15
60 LET First_number%  = 100
70 LET Second_number%  = 20
80 REM --------------------------
90 LET P% = Start_of_program%
```
100 [ 
110 .prog 
120 
130 
140 
150 
160 ] 
170 REM ------------------------
180 INPUT "Press return "$A$
190 CALL prog
200 PRINT ;"The answer is "?$&70
210 END

In this case, because we have avoided any difficult numbers, memory location &70 will hold the correct result. You will find the overflow flag will be set in a very similar program. Try the following:

10 REM *********************
20 REM * Simple overflow subtraction *
30 REM *********************
40 MODE 7
50 DIM Start_of_program% 15
60 LET First_number% = 130 :REM 
    (-126 In 2s complement)
70 LET Second_number% = 120
80 REM ------------------------
90 FOR Pass = 0 TO 3 STEP 3
100 LET P% = Start_of_program%
110 [ 
120 .prog 
130 
140 
150 
160 
170 .Text 
180 
190 
200 ]
210 NEXT
220 REM ----------------------
230 INPUT "Press return "A$
240 CALL prog
250 END

In Line 60, I have 'created' a two's complement number. The rest is very similar to the addition program. If you change the first number to +126, then the overflow flag will not be set.

SUMMARY

ADC ADd number + accumulator + Carry flag
SBC SuBtract number from accumulator, and also the Carry flag
CLC CLear the Carry flag and reset it to zero
SEC SEt the Carry flag to 1
BVS Branch on oVerflow being Set to 1
BVC Branch on oVerflow being Clear (0)

The other branch instructions are:

BCS Branch if the Carry flag is Set to 1
BCC Branch if the Carry flag is Clear (0)
BPL Branch if the sign flag is Positive (reset to zero)
BMI Branch if the sign flag is negative (set to 1)

The sign flag in the P register is affected by any instruction which will change the MSB of one of the 6502's registers - ie the extreme left-hand bit of a register.

Similarly the carry flag in the P register is set to 1 or reset to 0 depending on whether there is a ninth bit being carried.

Detailed description of how to use all the branch instructions successfully is beyond the scope of this book. But you should now know enough to experiment and learn for yourself: so good luck!
CHAPTER 11
OTHER ADDRESSING MODES

Other addressing modes

So far you have come across the following five different ways of accessing registers and memory addresses.

1. Immediate addressing

This means loading a 6502 register with actual data. Immediate addressing is shown by using the ‘#’ sign before the actual data you wish to load, eg

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA #65</td>
<td>put 65 into the accumulator</td>
</tr>
<tr>
<td>CMP #&amp;41</td>
<td>compare accumulator with the actual hex number &amp;41</td>
</tr>
</tbody>
</table>

2. Absolute addressing

This means accessing directly one of the memory locations in the memory chips of your computer - AND doing something with (or to) the contents of that memory location, eg

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC &amp;3000</td>
<td>Reduce by one the contents of memory</td>
</tr>
</tbody>
</table>
STA &5000 Store a copy of the accumulator in memory location &5000

3. Relative addressing

On the 6502 chip, this is only available with branch instructions. Which memory location you are to branch to is worked out as a displacement from the memory location where the branch instruction is stored. There is a maximum displacement from the end of the branch instruction of either +127 memory locations or -128 memory locations, eg

BNE &F7 go back 9 memory locations
BVS &09 go forward 9 memory locations

4. Zero page addressing

This is similar to absolute addressing except that, when accessing memory locations in the zero page, you can forget the high part of the address, eg

LDY &71 load Y from what is stored in memory location &0071
STA &80 store a copy of the accumulator in memory location &0080

5. Indexed addressing

This is used to access a memory address that is a certain number of memory locations away from a 'base' address. The X or Y registers can hold the displacement from the base address (except as described earlier). The index range is from 0 to 255 memory locations away from the base address.

If 8 is stored in the X register, then, eg
ADC &2000,X means add with carry the contents of memory location &2008
LDA &1500,X means load the accumulator from location &1508

INDIRECT ADDRESSING

This leaves the various indirect addressing modes. The 6502 chip is not as clever as some other microprocessors at indirect addressing. Two methods of indirect addressing are provided: indexed indirect addressing, and indirect indexed addressing. Sometimes these are described as pre-indexed and post-indexed addressing.

It is possible to use two consecutive memory locations to store the sixteen bits representing memory address. One can then access the two consecutive memory locations which will 'point' to the final address (see Figure 11).

Fig 11 USING 'INDIRECT' ADDRESSING
Indirect addressing is powerful because it allows the development of ‘relocatable code’. Any final memory addresses you will need can be stored in the pointer memory locations. So if the memory locations at which some of the machine code sits are changed, then all you need do is change the contents of the pointers.

The 6502 chip does not provide such a full facility as in the diagram, but nevertheless it does provide two useful versions.

1. **Indexed indirect addressing (pre-index)**

The index register X, when added to a zero page address, will access two consecutive eight-bit memory locations that will have stored in them the final 16-bit address. Figure 12 should help you to see how this works, eg

---

NOTE: Two zero page addresses - 76 hex and 77 hex - are needed for the 16 bit final address of the data.

---

**12 PRE-INDEX ADDRESSING**
LDA (&70, X) This means access zero page address &70. The &70 is now the base address for the X register. Add the contents of the X register to &70 to find two consecutive memory locations which will hold the 16-bit memory location which points to the final data.

2. Indirect indexed addressing (post-index)

In this method of indirect addressing, the zero page address alone provides the two consecutive memory locations with the final 16-bit address in them. Then the Y register provides an index from the indirect address. This indirect address provides a base address for the Y register. Figure 13 should help, e.g.

![Diagram of indirect indexed addressing]

NOTE: Two zero page memory locations are needed for the 16 bit final address of 3000 hex

Fig 13 POST-INDEX ADDRESSING
LDA (&80),Y This means access the zero page memory location &80. The 16-bit address in locations &80 and &81 will be the base address for the indexing by the Y register.

Notice that all indirect addresses are put into parentheses. This is standard for all assemblers.

The essential difference between these two kinds of indirect addressing is:

1. Using the X register for pre-index addressing allows you to set up a table of final indirect addresses in the zero page of memory. As the memory addresses which are being referred to indirectly need sixteen bits, the X register must step through the table of addresses in the zero page, in pairs of zero page addresses. You will need to use 'INX' followed again by 'INX' to step through the address table.

2. Using the Y register for post-index addressing allows you to keep a table of data anywhere in memory and to point to that table from a pair of zero page memory locations. The Y register can then index that table of data.

These final addressing modes are not easy to understand. I have mentioned them because when you look at another person's machine code you may not understand what is happening. It could provide a useful way of accessing a data table of relocatable graphics addresses, etc, or an address table for interrupt handling routines.

OTHER 6502 INSTRUCTIONS

It is not my intention to teach you 6502 machine code here. Other books will do that in more detail, and some suggested further reading is provided. However, though you can get started very nicely with the instructions and addressing modes I have described, you will come across the following instruction mnemonics as well.

THE 'LOGICAL' INSTRUCTIONS

AND

Any specified memory location (using a valid addressing mode) is 'ANDed' with the accumulator. Look at the following diagram:
Pretend that zero page location &70 has 65 decimal stored in it. Pretend that the accumulator has 90 decimal stored in it. A valid instruction would be ‘AND &70’.

01000001  (contents of &70)
01011010  (contents of accumulator)
-----------
01000000  (new contents of accumulator)

↑ The only bit in &70 and the accumulator that is the same.

Each bit in the accumulator is in turn compared with the specified address, and if there is a 1 in that position in both the accumulator and the specified memory location, then that bit in the accumulator will stay as a 1. Otherwise it will revert to a 0. It works in exactly the same manner as the AND instruction in BASIC.

The most common use of the AND instruction is as a ‘mask’. It allows you to spot if a certain bit in a register has been used. If you want to see if the MSB has been set to 1, you could perform the following instruction:

LDA #128
AND &70

If you get a zero result in the accumulator, it will be because the data in zero page &70 does not have a 1 in the extreme left-hand bit!

OR

Any specified memory location (using a valid addressing mode) is ‘OREd’ with the accumulator. Look at the following diagram:

Pretend that zero page location &70 has 65 decimal stored in it. Pretend that the accumulator has 90 decimal stored in it. A valid instruction would be ‘OR &70’.

01000001  (contents of &70)
01011010  (contents of accumulator)
-----------
01011011  (new contents of accumulator)

↑ ↑ ↑ ↑ Bits that are 1 in either &70 or the accumulator or both
Each bit in the accumulator is in turn compared with the specified address, and if there is a 1 in that position in either the accumulator or the specified memory location or both, then that bit in the accumulator will stay as a 1. Otherwise it will revert to a 0. It works in exactly the same manner as the OR instruction in BASIC.

**EOR**

Any specified memory location (using a valid addressing mode) is ‘Exclusive ORed’ with the accumulator. Look at the following diagram:

Pretend that zero page location &70 has 65 decimal stored in it. Pretend that the accumulator has 90 decimal stored in it. A valid instruction would be ‘EOR &70’.

\[
\begin{align*}
01000001 & \quad \text{(contents of } \&70) \\
01011010 & \quad \text{(contents of accumulator)} \\
00011011 & \quad \text{(new contents of accumulator)} \\
\uparrow \uparrow \uparrow & \quad \text{Bits that are 1 in either } \&70 \text{ or the accumulator, but not both.}
\end{align*}
\]

Each bit in the accumulator is in turn compared with the specified address, and if there is a 1 in that position in either the accumulator or the specified memory location, but not both, then that bit in the accumulator will stay as a 1. Otherwise it will
revert to a 0. It works in exactly the same manner as the EOR instruction in BASIC.

THE STACK INSTRUCTIONS

Earlier in this book I described how the 6502 stack operates. The 6502 chip provides a series of instructions for putting data on to the stack. There are two kinds of instruction that affect the stack.

Push/Pull

You may temporarily save both the accumulator in the stack and also all the flags in the P register. As so much data handling goes through the accumulator, you will find this temporary saving increasingly important. While doing an addition in the accumulator, you may want to use an OSWRCH routine. The stack would be an easiest temporary store for your addition.

When we put some data on to the stack we say that we are 'pushing' data on to the stack. To copy the accumulator to the stack we use the instruction PHA (PushH Accumulator). When we retrieve a copy of what was in the accumulator we say that we are 'pulling' data off the stack. We use the instruction: PLA (PullL Accumulator).

Similarly for the P register of flags PHP (PushH P register); PLP (PullL P register).

Return addresses

When you use the JSR command, the return address (sixteen bits) is automatically dumped as two bytes of memory on to the stack by the 6502 chip. In fact an oddity of the 6502 chip is that it places 'one less than the actual JSR return address' on to the stack. Similarly the RTS instruction automatically takes the top two bytes of memory off the top of the stack (in the hope that these are the return address). It then automatically adds one to this memory location to find the return memory address.

The problem is that you must be certain that you never leave on the stack any data that you don’t want. If you do, you may find that you get lost in that ‘pile of books’ we call the stack. Then it is possible that the return address for an RTS instruction will get mixed up, and your program will crash brilliantly.

The golden rule for the stack: pull off all you push on. (Do not use the stack as a data garbage bin.)
This will often involve you in a seeming waste of PLA and PLP instructions. Such care is never wasted.

Both the ‘REturn from Interrupt’ (REI) and ‘BReaK’ (BRK) instructions affect the stack.

**BRK**

This instruction interrupts the workings of the 6502 chip. It works in assembly language very similarly to the STOP command in BASIC. You may know that there is a bug in a section of your machine code. You don’t want the program to run away with itself, so at a suitable point in your assembly program you will insert a BRK instruction. When the processor meets the BRK instruction it will return you to BASIC so that you can edit your program.

For example, you may suspect that a branch instruction is secretly leading your program to an area of memory which you have used for something else. Put a BRK instruction into this part of memory, and if your program ‘bombs out’ and returns to BASIC, then you have confirmation that you have indeed wandered into the part of memory you suspected.

**THE SHIFT/ROTATE INSTRUCTIONS**

- **ROL**  ROtate Left
- **ROR**  ROtate Right
- **LSR**  Logical Shift Right
- **ASL**  Arithmetic Shift Left

**OTHER INSTRUCTIONS**

- **NOP**  Do nothing
- **CPX**  ComPare with contents of X
- **CPY**  ComPare with contents of Y
- **SEI**  SEt Interrupt disable
- **SED**  SEt Decimal mode
- **CLV**  CLean the oVerflow flag
- **CLI**  CLean the Interrupt disable flag
- **BIT**  Test a particular BIT for a 1 or 0

When you wish to handle assembler multiplication, then you will need the shift and rotate instructions. All these instructions are described in detail in 6502 manuals - and when you need to use them you will have finished with this book.
CHAPTER 12
MORE ADVANCED FEATURES OF ASSEMBLERS

Simple assemblers assemble everything that is in the ‘source’ program to machine code. Cleverer assemblers allow you to include machine code that you have already assembled earlier. When you get to be good at writing 6502 machine code on the BBC Micro, you will find that you have written lots of clever and useful ‘routines’. A routine will be a piece of code that does some useful task, like multiplying two numbers. You will not enjoy having to write these routines from scratch every time you need them. You could get quite irritated.

MACROS

The clever assemblers allow you to use ‘macros’. A macro is used like this.

When you are writing your source program, you may remember that you have earlier written a routine that displays a counter on the screen. You saved this routine on disc or tape and gave it a name. Now, instead of writing it again, you place the name of the routine into your source program in such a way that the assembler ‘recognises’ it as a macro. The assembler then hunts through the disc or tape for the name of your routine, finds it and includes it automatically as part of your source program. This only occurs when you decide to assemble the program.

You could find that in the future your advanced assembler programs were only six or seven lines long, because each line calls a macro that you had already written.

Of course you need to define macros and save them to disc or tape. When you think that you have written a particularly useful piece of code, then you should also save it as a macro.

A macro operates in two ways: it can either include itself into your source program as assembler mnemonics or it can include itself as ‘readymade’ machine code. The former is much more common as it allows addresses, etc, to vary.

The BBC Micro cannot properly support the use of macros. There is, however, a useful half-way compromise. When you
write your BASIC programs, you will probably be using defined functions and procedures. To call these procedures into action you simply write 'PROCcounter' or 'FNpythag' or something similar. If you know that you are likely to use an assembler routine several times in one program, then define it as a procedure or function. Then you need only write it once. You can use the 'PROC' command in exactly the same way to call the section of machine code. You can use the PROC command to pass parameters into your procedure because some of the resident integer variables also hold a copy of some of the 6502 registers:

A% = the accumulator
X% = the X register
Y% = the Y register
C% (least significant bit) = carry flag

```
10 REM ****************************
20 REM * procedure driven assembler  *
30 REM ****************************
40 MODE 7
50 DIM Start% 30
60 P% = Start%
70 C% = 0:A% = 30
```
90 REM -----------------------------
100 [   /This shows that a procedure
110 OPT 3   /can be put in the middle
120 ]     /of an assembler program
130 PROCadd(A%,C%)
140 [
150 OPT 3
160   STA &70
170   RTS
180 ]
190 INPUT "Press return "A$
200 CALL Start%
210 PRINT ?&70 "is the result."
220 END
230 DEF PROCadd(A%,C%)
240 [
250 OPT 3
260   CLC
270   ADC #45
280 ]
290 ENDPROC

Every section of assembler code for which you want an assembly listing must have a new OPT command. So OPT 3 has been repeated at Lines 110, 150 and 250. Any parameters can be passed into the routine, if you need them. Otherwise the code is very similar to the earlier addition program.

The listing you should see will look like this:

1AD1
1AD1    OPT 3
1AD1
1AD1    OPT 3
1AD1   18    CLC
1AD2 69 2D    ADC #45
1AD4
1AD4    OPT 3
1AD4 85 70    STA &70
1AD6 60

Press return 75 is the result.
You will notice some ‘blank’ lines such as for the ‘1AD1’s. This is what happens as one switches in and out of BASIC. Don’t worry: it has not affected the program. The machine code generated is: ‘18 69 2D 85 70 60’ and it all sits at consecutive memory locations ready for the program counter.

**CONDITIONAL ASSEMBLY**

There are times when you may wish to assemble different sections of an assembly program depending on some condition. For instance, if you have written a program for tape and disc machines, you may wish to sell two versions of the program. You will not want to write the whole program out twice with minor changes. The main difference between your tape and disc versions might be that they do their ‘saving’ in different ways.

The following code would be valid:

```plaintext
IF Type$ = "DISC" THEN PROCdisc__save ELSE PROCtape__save.
```

You will write two macros. Then you assemble your program. The first time you make ‘Type$’ equal to ‘DISC’, the second time you make it equal to ‘TAPE’.

This is called ‘conditional assembly’. All sorts of conditions can be used. You can even ‘cheat’ when using conditional assembly. If you are not certain that a ‘result’ needs sixteen or twenty four bits, you can write two routines and make either dependent on the value of result. I do not recommend this ‘lazy’ way of working as it is fraught with pitfalls.

Other features of assemblers that are a little cumbersome to get to work on the BBC Micro are ‘defined space’ and ‘defined message’. The first occurs when you wish to leave memory locations free for data in the middle of the assembly program. Many assemblers operate a ‘DEFS’ command which will leave empty memory locations. On the BBC Micro we have to reset the program counter by writing:

```
} 
LET P% = P% + 100
[
```
Remember we have to come out of assembler before resetting the P%.

The 'DEFM' command on many assemblers allows you to define the next twenty bytes or so as a text message. On the BBC Micro we get round this by using the string variables. Unfortunately there is no operating system call that easily mimics the BASIC PRINT command, when applied to strings.

THE CALL INSTRUCTION

It is possible to pass parameters through the CALL instruction. To do this memory locations from &600 are set aside as a parameter block. You do not need to reserve memory for the parameter block as the BBC Micro has done this for you.

In &600 you put the number of parameters that are being passed, and then the next three bytes, &601, &602 and &603 contain the address of the parameter (&601 and &602) and in the type of parameter in (&603). These three bytes are repeated for as many parameters as you have, eg

CALL &3000,Score%,Total%

will use up memory locations &600 to &606 in the parameter block. You will need to refer to Page 215 of the USER GUIDE for a more detailed description on the parameter types. I suggest that you do not use parameters in a CALL statement till you are feeling confident in your machine code.

SAVING MACHINE CODE PROGRAMS

Once you have assembled your program and it works (!), then you will not need or want continually to reassemble the program each time you use it. The *SAVE command is what you need.

If, when writing the assembly program, you use a command such as 'DIM Start% 50', then you know that the machine code starts at 'Start%' and finishes at 'Start% + 50'. Type:

PRINT ~Start%
PRINT ~50

A hexadecimal number such as 1AD1 will appear. The hex of decimal 50 will appear as 32. Then type:
*SAVE "Progname" 1AD1 +32

The format you have used is that the first hex number after the program name is the memory location where your machine code starts. The ‘+’ sign tells you how many memory locations to save. If you know that your program is not using the full 50 (decimal) memory locations reserved in the DIM Start% 50, then you can put after the ‘+’ sign the number of memory locations actually used.

All numbers with the *SAVE command go in hexadecimal.

You are not in the clear yet! Every time you use the *LOAD "Progname", the program will load and sit at the memory locations originally specified by Start%. This may not be what you either wanted or intended. If you type:

*LOAD "program name" 0D00

The program will then load into memory locations &0D00 upwards. The 4-digit hex number after *LOAD "program name" tells you where to load the program. Do remember that this address for machine code may cause you problems with a disc-based machine.

You can ‘chain’ a machine code program by using the ‘*RUN’ command. With the programs in this book that mix BASIC and assembler unashamedly that facility is not generally open to you. With *RUN the program counter is automatically set to the first memory location where your machine code has been loaded. It then executes (runs) the machine code from that address. There are times when you will not want this.

If you type:

*SAVE "Progname" SSSS +LLLL EEEE

then the EEEE is a 4-digit hex address which is the execute address of the program. When you *RUN what you have saved then it will execute from the execute address that you *SAVED.

This means that you can locate your machine code program anywhere in memory and execute it from any memory address. If you also write proper relocatable code, you will be writing useful general purpose programs.

The *LOAD facility allows you to write a short BASIC program which is a series of CALLs to machine code programs. This will dramatically shorten your BASIC program with its assembler content, eg
10 PRINT "Title"
20 *LOAD "Pic" &3000
30 *LOAD "Play" &4000
40 *LOAD "Finish" &5000
50 CALL &3000
60 REPEAT
70 CALL &4000
80 UNTIL FALSE
90 CALL &5000
100 END
CHAPTER 13
LONGER PROGRAMS

The following programs illustrate how to use some of the machine code techniques that you have read about so far. None of the programs can claim to be ‘useful’, but they can at least claim to be fun or amusing. I suggest that you leave out remarks, etc, when you type them in but stick to the same line numbers as me, because when you ‘de-bug’ you will have a reference listing in the book.

THE CHRISTMAS TREE PROGRAM

This program is probably better written in BASIC but it is a nice illustration of colour graphics. It does show that there are limitations to the speed of machine code graphics if you use the operating system routines. If you decide to by-pass ACORN’s routines and access the video chip directly, the program can be speeded slightly.

All you see is a pretty Christmas tree in green, with a blue base. The background is red and there are flashing cyan lights at the tips of the branches. There is a ‘Happy Christmas’ message at the end. Leave it in a window of your house and amuse your neighbours - but don’t tell them how long it took to type in!

Lines 10-140 The program works on the basis of continually changing the ‘y’ axis origin as the tree is drawn from the top down. The ‘y’ axis offset will be held on the stack.

Lines 150-330 The initial data for the first triangle of branches is passed from data statements to a reserved memory area for such data. As the ‘x’ and ‘y’ coordinates can be larger than 255, they must be split into low and high sections.

Lines 340-770 All the main details of the graphics are defined. The counter is the count of the number of Christmas tree branches. Note the redefining of the colours.

Lines 780-1210 A green triangle branch is drawn and then the
left and right tips of the branch have a cyan star added to them.

**Lines 1220-1450** The tree grows in size, like any fir tree, as it gets towards the base. To do this a value is added to the right tip and taken off the left tip of the branch.

**Lines 1450-2350** A series of tedious graphics calls to change the origin (VDU 29) and then to draw the base once fourteen branches have been drawn. The stack has to be cleared at 2310 in order to remove the counter and the 'y' axis details which it had.

**Lines 2380-3080** A series of graphics routines which are exact copies of what they would have been in BASIC.

**Lines 3080-end** More graphics routines which are part of the flashing star.

**PROGRAM LISTING**

```
10 REM ***********************************************
20 REM ***** Written by Ian Clarke. *****
30 REM ***** July 83 *****
40 REM ***********************************************
50
60 REM =-----------------------------------------------
70 REM set up constants for the
80 REM program to use
90 REM . . . . . . . . .
100
110 Y_axis_offset% = 900
120 OSWRCH = &FFEE
130 DIM Space% 600
140
150 REM =-----------------------------------------------
160 REM place the initial tree data
170 REM into the first twelve memory
180 REM locations reserved for m/code
190 REM . . . . . . . .
200
210 FOR X = 0 TO 10 STEP 2
```
220 READ Tree_data
230 LET ?(Space%+X) = Tree_data MOD 256
240 LET ?(Space%+X+1) = Tree_data DIV 256
250 NEXT X
260
270 REM =--------------------------------------------------
280 REM data is held as pairs of 'X'
290 REM and 'Y' coordinates
300 REM ................
310
320 DATA 600,90,660,0,540,0
330
340 REM =--------------------------------------------------
350 REM start of the machine code program
360 REM ................
370
380 FOR Pass% = 0 TO 3 STEP 3
390 P% = Space% + 13
400 |
410 OPT Pass%
420
430 / SET TREE BRANCH COUNTER TO ZERO
440 / AND PLACE COUNTER ON THE STACK
450
460 LDA #0
470 PHA
480 LDA #Y_axis_offset% DIV 256
490 PHA
500 LDA #Y_axis_offset% MOD 256
510 PHA
520
530 / SET UP MODE 5 GRAPHICS
540
550 LDA #22
560 JSR OSWRCH
570 LDA #5
580 JSR OSWRCH
590
600 / SET UP NEW LOGICAL COLOURS
610 / COLOUR 0 BECOMES RED
620 / COLOUR 1 BECOMES BLUE
630 / COLOUR 2 BECOMES FLASHING RED/CYAN
640 / COLOUR 3 BECOMES GREEN
650
660   LDX #2
670   LDY #9
680   JSR Logic_colour%
690   LDX #0
700   LDY #1
710   JSR Logic_colour%
720   LDX #3
730   LDY #2
740   JSR Logic_colour%
750   LDX #1
760   LDY #4
770   JSR Logic_colour%
780   .Main_program% LDY #3
790
800 / ==--------------------------------------
810 / THE MAIN PROGRAM
820 / THE TREE IS DRAWN FROM TOP DOWN
830 / SUB-ROUTINES ARE CALLED FROM
840 / TO PERFORM SPECIFIC TASKS
850 / -------------------------------
860
870   JSR Plot_colour%
880   JSR Graph_origin%
890
900 / START BY MOVING TO TOP AND RIGHT
910 / HAND POINTS OF TREE TRIANGLE
920
930   LDX #0
940   .Plot_4__call% JSR Point%
950   .Left_and_top% LDA Space%,X
960   JSR OSWRCH
970   INX
980   CPX #4
990   BEQ Plot_4__call%
1000   CPX #8
1010   BNE Left_and_top%
1020
1030 / NOW COMPLETE TRIANGLE BY DRAWING
1040 / TO LEFT HAND SIDE AS WELL
1050
1060 JSR Triangle%
1070 .Left__side% LDA Space%,X
1080 JSR OSWRCH
1090 INX
1100 CPX #12
1110 BNE Left__side%
1120
1130 / THE STAR AT THE END OF EACH BRANCH
1140 / IS DRAWN SIMILARLY TO THE MAIN
1150 / BRANCH TRIANGLE
1160
1170 LDX #4
1180 JSR Star%
1190 LDX #8
1200 JSR Star%
1210
1220 / MAKE THE TREE GROW BY ADDING AN
1230 / OFFSET TO THE TREE TRIANGLES.
1240 / FIRST THE RIGHT SIDE BY ADDING THE OFFSET
1250
1260 LDA Space%+4
1270 CLC
1280 ADC #30
1290 STA Space%+4
1300 TAY
1310 BCC Left__offset%
1320 LDY Space%+5
1330 INY
1340 STY Space%+5
1350
1360 / NOW THE LEFT SIDE BY SUBTRACTING
1370 / THE OFFSET
1380
1390 .Left__offset% LDA Space%+8
1400 SEC
1410 SBC #30
1420 STA Space%+8
1430 TAY
1440 BCS Change__origin%
1450
1460 / MOVE THE TREE DRAWING ORIGIN LOWER
1470 / READY FOR THE NEXT TREE TRIANGLE
1480    LDY Space%+9
1500    DEY
1510    STY Space%+9
1520    .Change__origin% PLA
1530    TAX
1540    PLA
1550    TAY
1560    TXA
1570    SEC
1580    SBC #60
1590    TAX
1600    BCS End__check%
1610    DEY
1620
1630 / CHECK THAT JUST 14 BRANCHES OF
1640 / THE TREE ARE BEING DRAWN
1650
1660    .End__check%    PLA
1670    CLC
1680    ADC #1
1690    STA Space%+12
1700    PHA
1710    TYA
1720    PHA
1730    TXA
1740    PHA
1750    LDA Space%+12
1760    CMP #14
1770    BEQ Tree__base%
1780    JMP Main__program%
1790
1800 / DRAW THE BASE OF THE TREE AS A
1810 / SERIES OF TRIANGLES
1820
1830    .Tree__base%    JSR Graph__origin%
1840    LDY #1
1850    JSR Plot__colour%
1860    JSR Point%
1870    LDA #500 MOD 256
1880    JSR OSWRCH
1890    LDA #500 DIV 256
| 1900 | JSR OSWRCH       |
| 1910 | LDA #0          |
| 1920 | JSR OSWRCH       |
| 1930 | JSR OSWRCH       |
| 1940 | JSR Point%       |
| 1950 | LDA #430 MOD 256 |
| 1960 | JSR OSWRCH       |
| 1970 | LDA #430 DIV 256 |
| 1980 | JSR OSWRCH       |
| 1990 | LDA #60          |
| 2000 | JSR OSWRCH       |
| 2010 | LDA #0           |
| 2020 | JSR OSWRCH       |
| 2030 | JSR Triangle%    |
| 2040 | LDA #770 MOD 256 |
| 2050 | JSR OSWRCH       |
| 2060 | LDA #770 DIV 256 |
| 2070 | JSR OSWRCH       |
| 2080 | LDA #60          |
| 2090 | JSR OSWRCH       |
| 2100 | LDA #0           |
| 2110 | JSR OSWRCH       |
| 2120 |                 |

2130 / SECOND TREE BASE TRIANGLE

| 2140 |                 |
| 2150 | JSR Point%      |
| 2160 | LDA #700 MOD 256 |
| 2170 | JSR OSWRCH      |
| 2180 | LDA #700 DIV 256 |
| 2190 | JSR OSWRCH      |
| 2200 | LDA #0          |
| 2210 | JSR OSWRCH      |
| 2220 | JSR OSWRCH      |
| 2230 | JSR Triangle%   |
| 2240 | LDA #500 MOD 256 |
| 2250 | JSR OSWRCH      |
| 2260 | LDA #500 DIV 256 |
| 2270 | JSR OSWRCH      |
| 2280 | LDA #0          |
| 2290 | JSR OSWRCH      |
| 2300 | JSR OSWRCH      |
| 2310 | PLA             |
2320 PLA
2330 PLA
2340 RTS
2350
2360 / END OF MAIN PROGRAM
2370 / ........................................
2380
2390 / SUBROUTINES FOLLOW
2400
2410 .Point% LDA #25
2420 JSR OSWRCH
2430 LDA #4
2440 JSR OSWRCH
2450 RTS
2460
2470 .Triangle% LDA #25
2480 JSR OSWRCH
2490 LDA #85
2500 JSR OSWRCH
2510 RTS
2520
2530 .Graph__origin% PLA
2540 TAX
2550 PLA
2560 TAY
2570 LDA #29
2580 JSR OSWRCH
2590 LDA #0
2600 JSR OSWRCH
2610 JSR OSWRCH
2620 PLA
2630 JSR OSWRCH
2640 STA Space%+12
2650 PLA
2660 JSR OSWRCH
2670 PHA
2680 LDA Space%+12
2690 PHA
2700 TYA /(Transfer the program
2710 PHA /counter back to the stack.)
2720 TXA
2730 PHA
2740 RTS
2750
2760 .Logic__colour%  LDA #19
2770 JSR OSWRCH
2780 TXA
2790 JSR OSWRCH
2800 TYA
2810 JSR OSWRCH
2820 LDA #0
2830 JSR OSWRCH
2840 JSR OSWRCH
2850 JSR OSWRCH
2860 RTS
2870
2880 .Plot__colour%  LDA #18
2890 JSR OSWRCH
2900 LDA #0
2910 JSR OSWRCH
2920 TYA
2930 JSR OSWRCH
2940 RTS
2950
2960 .Star%    LDY #2
2970 JSR Plot__colour%
2980 JSR Point%
2990 LDA Space%\,X
3000 JSR OSWRCH
3010 LDA Space%+1\,X
3020 JSR OSWRCH
3030 LDA Space%+2\,X
3040 JSR OSWRCH
3050 LDA Space%+3\,X
3060 JSR OSWRCH
3070 JSR Point%
3080
3090 / RIGHT SIDE OF STAR
3100
3110 LDY Space%+1\,X
3120 LDA Space%\,X
3130 CLC
3140 ADC #13
3150 PHP /(Stack status register since the
JSR OSWRCH / next call alters it.)
PLP
BCC High__axis__Y1%
INY
.JHigh__axis__Y1% TYA
JSR OSWRCH
LDY Space%+3,X
LDA Space%+2,X
SEC
SBC #13
PHA /(Stack low byte off Y__axis.)
PHP /(Stack status register since the
JSR OSWRCH / next call alters it.)
PLP
BCS High__axis__Y2%
DEY
.JHigh__axis__Y2% TYA
PHA /(Stack high byte off Y__axis.)
JSR OSWRCH
/ DRAW LEFT SIDE OF TRIANGLE
JSR Triangle%
LDY Space%+1,X
LDA Space%,X
SEC
SBC #13
PHP
JSR OSWRCH
PLP
BCS High__X__axis%
DEY
.JHigh__X__axis% TYA
JSR OSWRCH
PLA /(Unstack high byte off Y__axis.)
TAY
PLA /(Unstack low byte off Y__axis.)
JSR OSWRCH
TYA
JSR OSWRCH
RTS
3580 NEXT Pass%
3590
3600 REM -----------------------------------------------
3610 REM activate machine code program
3620 REM .................
3630
3640 PRINT
3650 INPUT "When ready press the RETURN key "A$
3660 CALL Space%+13
3670 PRINT TAB(1,31);"Happy Christmas.";
3680
>RUN
40A9
40A9
40A9
40A9
40A9
40A9
40A9 A9 00
40AB 48
40AC A9 03
40AE 48
40AF A9 84
40B1 48
40B2
40B2
40B2 A9 16
40B4 20 EE FF
40B7 A9 05
40B9 20 EE FF
40BC
40BC
40BC
40BC
40BC
40BC
40BC
40BC
OPT Pass%
/ SET TREE BRANCH COUNTER TO ZERO
/ AND PLACE COUNTER ON THE STACK
LDA #0
PHA
LDA #Y_axis_offset% DIV 256
PHA
LDA #Y_axis_offset% MOD 256
PHA
/ SET UP MODE 5 GRAPHICS
LDA #22
JSR OSWRCH
LDA #5
JSR OSWRCH
/ SET UP NEW LOGICAL COLOURS
/ COL 0 BECOMES RED
/ COL 1 BECOMES BLUE
/ COL 2 BECOMES FLASHING RED/ CYAN
/ COL 3 BECOMES BECOMES GREEN
40BC A2 02  LDX #2
40BE A0 09  LDY #9
40C0 20 09 42  JSR Logic__colour%
40C3 A2 00  LDX #0
40C5 A0 01  LDY #1
40C7 20 09 42  JSR Logic__colour%
40CA A2 03  LDX #3
40CC A0 02  LDY #2
40CE 20 09 42  JSR Logic__colour%
40D1 A2 01  LDX #1
40D3 A0 04  LDY #4
40D5 20 09 42  JSR Logic__colour%
40D8 A0 03  .Main__program% LDY #3
40DA
40DA  / ================================
40DA  / THE MAIN PROGRAM
40DA  / TREE IS DRAWN FROM TOP DOWN
40DA  / SUB-ROUTINES ARE CALLED FROM
40DA  / TO PERFORM SPECIFIC TASKS
40DA  / ....................................
40DA
40DA 20 22 42  JSR Plot__colour%
40DD 20 E3 41  JSR Graph__origin%
40E0
40E0  / START AT TOP AND RIGHT
40E0  / HAND POINTS OF TREE TRIANGLE
40E0
40E0 A2 00  LDX #0
40E2 20 CD 41  .Plot__4__call% JSR Point%
40E5 BD 9C 40  .Left__and__top% LDA Space%,X
40E8 20 EE FF  JSR OSWRCH
40EB E8  INX
40EC E0 04  CPX #4
40EE F0 F2  BEQ Plot__4__call%
40F0 E0 08  CPX #8
40F2 D0 F1  BNE Left__and__top%
40F4
40F4  / COMPLETE TRIANGLE BY DRAWING
40F4  / TO LEFT HAND SIDE AS WELL
40F4
40F4 20 D8 41  JSR Triangle%
40F7 BD 9C 40  .Left__side% LDA Space%,X
40FA 20 EE FF  JSR OSWRCH
40FD E8  INX
40FE E0 0C  CPX #12
4100 D0 F5  BNE Left__side%
4102
4102  /STAR AT THE END OF EACH BRANCH
4102  / IS DRAWN SIMILARLY TO THE MAIN
4102  / BRANCH TRIANGLE
4102
4102 A2 04  LDX #4
4104 20 31 42  JSR Star%
4107 A2 08  LDX #8
4109 20 31 42  JSR Star%
410C
410C  /MAKE TREE GROW BY ADDING AN
410C  /OFFSET TO THE TREE TRIANGLES.
410C  /FIRST ADD OFFSET TO RIGHT SIDE
410C
410C AD A0 40  LDA Space%+4
410F 18  CLC
4110 69 1E  ADC #30
4112 8D A0 40  STA Space%+4
4115 A8  TAY
4116 90 07  BCC Left__offset%
4118 AC A1 40  LDY Space%+5
411B C8  INY
411C 8C A1 40  STY Space%+5
411F
411F  /NOW THE LEFT SIDE BY
411F  /SUBTRACTING THE OFFSET
411F
411F AD A4 40  .Left__offset%  LDA Space%+8
4122 38  SEC
4123 E9 1E  SBC #30
4125 8D A4 40  STA Space%+8
4128 A8  TAY
4129 B0 07  BCS Change__origin%
412B
412B  /MOVE TREE DRAWING ORIGIN
412B  /LOWER READY FOR NEXT TRIANGLE
412B
412B AC A5 40  LDY Space%+9
412E 88 DEY
412F 8C A5 40 STY Space%+9
4132 68 .Change__origin% PLA
4133 AA TAX
4134 68 PLA
4135 A8 TAY
4136 8A TXA
4137 38 SEC
4138 E9 3C SBC #60
413A AA TAX
413B B0 01 BCS End__check%
413D 88 DEY
413E
413E / CHECK THAT JUST 14 BRANCHES OF
413E / THE TREE ARE BEING DRAWN
413E
413E 68 .End__check% PLA
413F 18 CLC
4140 69 01 ADC #1
4142 8D A8 40 STA Space%+12
4145 48 PHA
4146 98 TYA
4147 48 PHA
4148 8A TXA
4149 48 PHA
414A AD A8 40 LDA Space%+12
414D C9 0E CMP #14
414F F0 03 BEQ Tree__base%
4151 4C D8 40 JMP Main__program%
4154
4154 / DRAW THE BASE OF THE TREE AS A
4154 / SERIES OF TRIANGLES
4154
4154 20 E3 41 .Tree__base% JSR Graph__origin%
4157 A0 01 LDY #1
4159 20 22 42 JSR Plot__colour%
415C 20 CD 41 JSR Point%
415F A9 F4 LDA #500 MOD 256
4161 20 EE FF JSR OSWRCH
4164 A9 01 LDA #500 DIV 256
4166 20 EE FF JSR OSWRCH
4169 A9 00 LDA #0

138
416B 20 EE FF  JSR OSWRCH
416E 20 EE FF  JSR OSWRCH
4171 20 CD 41  JSR Point%
4174 A9 AE  LDA #430 MOD 256
4176 20 EE FF  JSR OSWRCH
4179 A9 01  LDA #430 DIV 256
417B 20 EE FF  JSR OSWRCH
417E A9 3C  LDA #60
4180 20 EE FF  JSR OSWRCH
4183 A9 00  LDA #0
4185 20 EE FF  JSR OSWRCH
4188 20 D8 41  JSR Triangle%
418B A9 02  LDA #770 MOD 256
418D 20 EE FF  JSR OSWRCH
4190 A9 03  LDA #770 DIV 256
4192 20 EE FF  JSR OSWRCH
4195 A9 3C  LDA #60
4197 20 EE FF  JSR OSWRCH
419A A9 00  LDA #0
419C 20 EE FF  JSR OSWRCH
419F
419F  / SECOND TREE BASE TRIANGLE
419F
419F 20 CD 41  JSR Point%
41A2 A9 BC  LDA #700 MOD 256
41A4 20 EE FF  JSR OSWRCH
41A7 A9 02  LDA #700 DIV 256
41A9 20 EE FF  JSR OSWRCH
41AC A9 00  LDA #0
41AE 20 EE FF  JSR OSWRCH
41B1 20 EE FF  JSR OSWRCH
41B4 20 D8 41  JSR Triangle%
41B7 A9 F4  LDA #500 MOD 256
41B9 20 EE FF  JSR OSWRCH
41BC A9 01  LDA #500 DIV 256
41BE 20 EE FF  JSR OSWRCH
41C1 A9 00  LDA #0
41C3 20 EE FF  JSR OSWRCH
41C6 20 EE FF  JSR OSWRCH
41C9 68  PLA
41CA 68  PLA
41CB 68  PLA
41CC 60 RTS
41CD
41CD / END OF MAIN PROGRAM
41CD / ..............................................
41CD
41CD / SUBROUTINES FOLLOW
41CD
41CD A9 19 .Point% LDA #25
41CF 20 EE FF JSR OSWRCH
41D2 A9 04 LDA #4
41D4 20 EE FF JSR OSWRCH
41D7 60 RTS
41D8
41D8 A9 19 .Triangle% LDA #25
41DA 20 EE FF JSR OSWRCH
41DD A9 55 LDA #85
41DF 20 EE FF JSR OSWRCH
41E2 60 RTS
41E3
41E3 68 .Graph__origin% PLA
41E4 AA TAX
41E5 68 PLA
41E6 A8 TAY
41E7 A9 1D LDA #29
41E9 20 EE FF JSR OSWRCH
41EC A9 00 LDA #0
41EE 20 EE FF JSR OSWRCH
41F1 20 EE FF JSR OSWRCH
41F4 68 PLA
41F5 20 EE FF JSR OSWRCH
41F8 8D A8 40 STA Space%+12
41FB 68 PLA
41FC 20 EE FF JSR OSWRCH
41FF 48 PHA
4200 AD A8 40 LDA Space%+12
4203 48 PHA
4204 98 TYA /(Transfer the program counter
4205 48 PHA / back to the stack.)
4206 8A TXA
4207 48 PHA
4208 60 RTS
4209
4209 A9 13 .Logic_colour% LDA #19
420B 20 EE FF JSR OSWRCH
420E 8A TXA
420F 20 EE FF JSR OSWRCH
4212 98 TYA
4213 20 EE FF JSR OSWRCH
4216 A9 00 LDA #0
4218 20 EE FF JSR OSWRCH
421B 20 EE FF JSR OSWRCH
421E 20 EE FF JSR OSWRCH
4221 60 RTS
4222
4222 A9 12 .Plot_colour% LDA #18
4224 20 EE FF JSR OSWRCH
4227 A9 00 LDA #0
4229 20 EE FF JSR OSWRCH
422C 98 TYA
422D 20 EE FF JSR OSWRCH
4230 60 RTS
4231
4231 A0 02 .Star% LDY #2
4233 20 22 42 JSR Plot_colour%
4236 20 CD 41 JSR Point%
4239 BD 9C 40 LDA Space%,X
423C 20 EE FF JSR OSWRCH
423F BD 9D 40 LDA Space%+1,X
4242 BD 9E 40 LDA Space%+2,X
4245 BD 9F 40 LDA Space%+3,X
4248 20 EE FF JSR OSWRCH
424B BD 9D 40 LDA Space%+1,X
4251 20 CD 41 JSR Point%
4254
4254 / RIGHT SIDE OF STAR
4254
4254 BC 9D 40 LDY Space%+1,X
4257 BD 9C 40 LDA Space%,X
425A 18 CLC
425B 69 0D ADC #13
425D 08 PHP /(Stack status register since the next
425E 20 EE FF JSR OSWRCH / call alters it.)
4261 28 PLP
4262 90 01  BCC High_axis_Y1%
4264 C8     INY
4265 98     .High_axis_Y1% TYA
4266 20 EE FF JSR OSWRCH
4269 BC 9F 40 LDY Space%+3,X
426C BD 9E 40 LDA Space%+2,X
426F 38     SEC
4270 E9 0D   SBC #13
4272 48     PHA /(Stack low byte off Y_axis.)
4273 08     PHP /(Stack status register since the next
4274 20 EE FF JSR OSWRCH / call alters it.)
4277 28     PLP
4278 B0 01   BCS High_axis_Y2%
427A 88     DEY
427B 98     .High_axis_Y2% TYA
427C 48     PHA /(Stack high byte off Y_axis.)
427D 20 EE FF JSR OSWRCH

4280
4280     / DRAW LEFT SIDE OF TRIANGLE
        4280
4280 20 D8 41  JSR Triangle%
4283 BC 9D 40  LDY Space%+1,X
4286 BD 9C 40  LDA Space%,X
4289 38     SEC
428A E9 0D   SBC #13
428C 08     PHP
428D 20 EE FF JSR OSWRCH
4290 28     PLP
4291 B0 01   BCS High_X_axis%
4293 88     DEY
4294 98     .High_X_axis% TYA
4295 20 EE FF JSR OSWRCH
4298 68     PLA /(Unstack high byte off Y_axis.)
4299 A8     TAY
429A 68     PLA /(Unstack low byte off Y_axis.)
429B 20 EE FF JSR OSWRCH
429E 98     TYA
429F 20 EE FF JSR OSWRCH
42A2 60     RTS

When ready press the RETURN key
THE REAL TIME CLOCK

This clever bit of code keeps a digital clock on display as long as you are using MODE 7. With a disc system you cannot use the saving or loading commands while the clock is functional as it has been assembled into some of the disc workspace, but you should find cassette saving unaffected. If there are any problems put the start of the program to &C00. Whatever you do—clear the screen, list or type in a program— the yellow digital clock will appear in the top right hand corner of the screen. Do remember to turn off the clock with *FX 13,4 or <BREAK> before running a BASIC program— or else havoc will be let loose (particularly if any other MODE is used).

Lines 10-360 Because one of the ‘events’ recorded in the computer is the VDU Vertical Synch Pulse, and because this ‘event’ occurs regularly every 1/50th of a second, it is possible to use this event to provide a reasonably accurate clock. It is more convenient to do this than to access the counter/timer and its parameter block, despite the fact that the counter/timer is also an ‘event’ detected by the computer. Each of the clock details (hours, minutes and seconds) are held in two memory locations. Centi (line 350) merely counts the number of times there has been an ‘event’ and the number of interrupt routines that have been triggered.

Lines 370-630 The start data for the clock is collected from the keyboard. It is a 24 hour clock but it could easily be reconfigured to be a 100 hour clock, etc.

Lines 640-830 There is a vertical synch pulse fifty times a second. This ‘event’ is recorded by the BBC Micro and can be used to trigger an interrupt. Once the processor has been interrupted it looks at this program— and these lines. In line 800 we wait for the fiftieth pulse; otherwise we return to BASIC with the RTS at line 820.

Lines 870-1270 These routines are accessed on the fiftieth pulse. First we add to the second counter. It can, of course, spill into the high half of the second counter or into the minute and hour counter. All the routines here are essentially the same.

Lines 1280-1620 At line 1360 we look for a ‘4’. This could mean we are at the 24:00:00 time when we need to reset
everything to 00:00:00. So at line 1530 we check to see if we also have a ‘2’. The time could of course be 14:00:00, in which case we do not need to reset everything. The clock is completely reset at line 1590.

**Lines 1640-1780** The BBC Micro keeps a record of its ‘hardware’ scrolling of the screen and puts into two memory addresses (&350 and &351) the address of the current start of the screen. This is really only of use to us in MODE 7 as we can write a character to a screen address. In other MODEs we can only affect a pixel (dot) of light. If you run this program in any other MODE you will see what I mean. We copy the start of screen down to zero page as only in zero page can we use indirect addressing. The &1C in line 1770 moves the clock to the right hand end of the top line of the screen.

**Lines 1790-2090** Directly accessing the screen, we use indexed indirect addressing to display the clock. Change the colour by changing the value in line 1790.

**Lines 2100-2230** The addresses here are where the BBC Micro wants you to put the address of your machine code program which will be executed if an ‘event’ interrupts the processor.

**Lines 2240-end** Obvious.

---

**PROGRAM LISTING**

```plaintext
10 REM ***************************************************
20 REM * REAL TIME CLOCK *
30 REM * *
40 REM * WRITTEN FOR THE *
50 REM * BBC-MICRO *
60 REM * WITH 1.2 OS ROM *
70 REM * BY *
80 REM * ANDREW PUSEY. AUGUST 83 *
90 REM ***************************************************
100
110 REM ================================================
120 REM disable the interrupt request
```
130 REM as re-running of the program
140 REM can cause the ‘event-handling’
150 REM to get confused
160 REM
170
180 *FX13,4
190 CLS
200
210 REM ================
220 REM allocate zero page addresses
230 REM for variables and set up the
240 REM program counter
250 REM
260
270 LET Start_of_program = &D00 /Use &C00 for disk
280 LET Ascii_offset = 48
290 LET High_second = &73
300 LET Low_second = &74
310 LET High_minute = &75
320 LET Low_minute = &76
330 LET High_hour = &77
340 LET Low_hour = &78
350 LET Centi = &79
360
370 REM ================
380 REM ask the time for the clock
390 REM and validate the entry.
400 REM Then work out start values
410 REM for the zero page stores.
420 REM
430
440 PRINT CHR$(131) "Enter times for clock"
450 PRINT CHR$(131) "NO DECIMAL VALUES PLEASE"
460 PRINT
470 PRINT CHR$(134) "Hours" CHR$(131);
480 INPUT H%
490 IF H% < 0 OR H% > 23 THEN PRINT CHR$(129)
   "0 TO 23 ONLY - redo": GOTO 470
500 PRINT CHR$(134) "Minutes " CHR$(131); 
510 INPUT M%
520 IF M% < 0 OR M% > 59 THEN PRINT CHR$(129)
   "0 TO 59 ONLY - redo": GOTO 500
530 PRINT CHR$(134) "Seconds " CHR$(131);
540 INPUT S%
550 IF S% < 0 OR S% > 59 THEN PRINT CHR$(129)
   "0 TO 59 ONLY - redo": GOTO 530
560
570 LET ?High_second = (S%/10) + Ascii_offset
580 LET ?Low_second = (S%-INT(S%/10)*10) +
                  Ascii_offset
590 LET ?High_minute = (M%/10) + Ascii_offset
600 LET ?Low_minute = (M%-INT(M%/10)*10) +
                  Ascii_offset
610 LET ?High_hour = (H%/10) + Ascii_offset
620 LET ?Low_hour = (H%-INT(H%/10)*10) +
                  Ascii_offset
630
640 REM -------------------------------
650 REM start of the assembly
660 REM language program
670 REM ............................
680
690 FOR Pass = 0 TO 3 STEP 3
700 LET P% = Start_of_program
710 [
720 OPT Pass
730
740 / WAIT FOR 50 INTERRUPTS BEFORE
750 / INCREMENTING SECOND COUNTER.
760
770 JSR DSPLY
780 INC Centi
790 LDA Centi
800 CMP #50
810 BEQ JUMP1
820 RTS
830
840 / WAIT FOR 10 SECONDS BEFORE
850 / INCREMENTING HIGH BYTE
860
870 JUMP1 LDA #00
880 STA Centi
890 INC Low_second
900 LDA Low_second
910  CMP #10 + Ascii_offset
920  BEQ JUMP2
930  RTS
940
950 / WAIT FOR 6 INCREMENTS OF HIGH
960 / BYTE BEFORE INCREMENTING MINUTE
970
980.JUMP2  LDA #00 + Ascii_offset
990  STA Low_second
1000  INC High_second
1010  LDA High_second
1020  CMP #06 + Ascii_offset
1030  BEQ JUMP3
1040  RTS 1050
1060 / WAIT FOR 10 MINUTES BEFORE
1070 / INCREMENTING HIGH BYTE
1080
1090.JUMP3  LDA #00 + Ascii_offset
1100  STA High_second
1110  INC Low_minute
1120  LDA Low_minute
1130  CMP #10 + Ascii_offset
1140  BEQ JUMP4
1150  RTS
1160
1170 / WAIT FOR 6 INCREMENTS OF HIGH
1180 / BYTE BEFORE INCREMENTING HOUR
1190
1200.JUMP4  LDA #00 + Ascii_offset
1210  STA Low_minute
1220  INC High_minute
1230  LDA High_minute
1240  CMP #06 + Ascii_offset
1250  BEQ JUMP5
1260  RTS
1270
1280 / WAIT FOR 10 HOURS BEFORE
1290 / INCREMENTING LOW BYTE
1300 / AND CHECK THAT 24 HRS IS NOT UP
1310
1320.JUMP5  LDA #00 + Ascii_offset
1330  STA High_minute
1340 INC Low__hour
1350 LDA Low__hour
1360 CMP #4 + Ascii__offset
1370 BEQ CHECK
1380 CMP #10
1390 BEQ JUMP6
1400 RTS
1410
1420 / WAIT FOR 2 HIGH BYTES BEFORE
1430 / RE-SETTING TIMER
1440
1450 JUMP6 LDA #00 + Ascii__offset
1460 STA Low__hour
1470 INC High__hour
1480 RTS
1490
1500 / SEE IF 24 HRS IS UP
1510
1520 CHECK LDA High__hour
1530 CMP #2 + Ascii__offset
1540 BEQ JUMP7
1550 RTS
1560
1570 / RESET HOURS TO ZERO
1580
1590 JUMP7 LDA #0 + Ascii__offset
1600 STA High__hour
1610 STA Low__hour
1620 RTS
1630
1640 / DISPLAY TIME ON THE MODE7 SCREEN
1650 / FIRST COLLECT START OF MODE7
1660 / SCREEN FROM &350 AND &351.
1670 / COPY THE ADDRESS TO A ZERO
1680 / PAGE ADDRESS FOR POST-INDEX
1690 / ADDRESSING. PUT INTO ‘Y’ AN
1700 / OFFSET TO MOVE DISPLAY ACROSS
1710 / TO RIGHT HAND SIDE OF SCREEN
1720
1730 DISPLAY LDA &350
1740 STA &70
1750 LDA &351
1760 STA &71
1770 LDY #&1C
1780
1790 LDA #131 /YELLOW COLOUR
1800 STA (&70),Y
1810 INY
1820 LDA High__hour
1830 STA (&70),Y
1840 LDA Low__hour
1850 INY
1860 STA (&70),Y
1870 LDA #58 / Ascii code for colon
1880 INY
1890 STA (&70),Y
1900 LDA High__minute
1910 INY
1920 STA (&70),Y
1930 LDA Low__minute
1940 INY
1950 STA (&70),Y
1960 LDA #58 / Ascii code for colon
1970 INY
1980 STA (&70),Y
1990 LDA High__second
2000 INY
2010 STA (&70),Y
2020 LDA Low__second
2030 INY
2040 STA (&70),Y
2050 INY
2060 LDA #135 / WHITE COLOUR
2070 STA (&70),Y
2080 RTS
2090 ]
2100 NEXT Pass
2110
2120 REM .........................
2130 REM finish of assembly program
2140 REM ==========================
2150
2160 REM ==========================
2170 REM set the IRQ vector to point
2180 REM to start of our clock program
2190 REM ..........................
2200
2210 LET ?&220 = Start_of_program MOD 256
2220 LET ?&221 = Start_of_program DIV 256
2230
2240 REM =..................................
2250 REM set function keys to turn
2260 REM on and off the IRQ event
2270 REM ..........................
2280
2290 *KEY8 *FX14,4 \M
2300 *KEY9 *FX13,4 \M
2310 CLS
2320 PRINT TAB(5,5) CHR$(131) "Function key 8 turns
2330 PRINT TAB(5) CHR$(131) "Function key 9 turns
2340 END
2350
>
RUN
Enter times for clock
NO DECIMAL VALUES PLEASE

Hours 23
Minutes 58
Seconds 59

0D00
0D00
0D00  OPT Pass
0D00
0D00 / WAIT FOR 50 INTERRUPTS BEFORE
0D00 / INCREMENTING SECOND COUNTER.
0D00
0D00 20 66 0D JSR DSPLAY
0D03 E6 79 INC Centi
0D05 A5 79 LDA Centi
0D07 C9 32 CMP #50
0D09 F0 01 BEQ JUMP1
0D0B 60 RTS
0D0C  / WAIT FOR 10 SECONDS BEFORE
0D0C  / INCREMENTING HIGH BYTE
0D0C
0D0C A9 00  .JUMP1  LDA #00
0D0C E8 57 9  STA Centi
0D0C E6 74  INC Low._second
0D0C A5 74  LDA Low._second
0D0C C9 3A  CMP #10 + Asciii__offset
0D0C F0 01  BEQ JUMP2
0D0C 60  RTS
0D19
0D19  / WAIT FOR 6 INCREMENTS OF HIGH
0D19  / BYTE BEFORE INCREMENTING MINUTE
0D19
0D19 A9 30  .JUMP2  LDA #00 + Asciii__offset
0D1B 85 74  STA Low._second
0D1D E6 73  INC High._second
0D1F A5 73  LDA High._second
0D21 C9 36  CMP #06 + Asciii__offset
0D23 F0 01  BEQ JUMP3
0D25 60  RTS
0D26
0D26  / WAIT FOR 10 MINUTES BEFORE
0D26  / INCREMENTING HIGH BYTE
0D26
0D26 A9 30  .JUMP3  LDA #00 + Asciii__offset
0D28 85 73  STA High._second
0D2A E6 76  INC Low._minute
0D2C A5 76  LDA Low._minute
0D2E C9 3A  CMP #10 + Asciii__offset
0D30 F0 01  BEQ JUMP4
0D32 60  RTS
0D33
0D33  / WAIT FOR INCREMENTS OF HIGH
0D33  / BYTE BEFORE INCREMENTING HOUR
0D33
0D33 A9 30  .JUMP4  LDA #00 + Asciii__offset
0D35 85 76  STA Low._minute
0D37 E6 75  INC High._minute
0D39 A5 75  LDA High._minute
0D3B C9 36  CMP #06 + Asciii__offset
0D3D F0 01 BEQ JUMP5
0D3F 60 RTS
0D40
0D40 / WAIT FOR 10 HOURS BEFORE
0D40 / INCREMENTING LOW BYTE
0D40 / AND CHECK THAT 24 HRS IS NOT UP
0D40
0D40 A9 30 .JUMP5 LDA #00 + Ascii_offset
0D42 85 75 STA High_minute
0D44 E6 78 INC Low_hour
0D46 A5 78 LDA Low_hour
0D48 C9 34 CMP #4 + Ascii_offset
0D4A F0 0C BEQ CHECK
0D4C C9 0A CMP #10
0D4E F0 01 BEQ JUMP6
0D50 60 RTS
0D51
0D51 / WAIT FOR 2 HIGH BYTES BEFORE
0D51 / RE-SETTING TIMER
0D51
0D51 A9 30 .JUMP6 LDA #00 + Ascii_offset
0D53 85 78 STA Low_hour
0D55 E6 77 INC High_hour
0D57 60 RTS
0D58
0D58 / SEE IF 24 HRS IS UP
0D58
0D58 A5 77 .CHECK LDA High_hour
0D5A C9 32 CMP #2 + Ascii_offset
0D5C F0 01 BEQ JUMP7
0D5E 60 RTS
0D5F
0D5F / RESET HOURS TO ZERO
0D5F
0D5F A9 30 .JUMP7 LDA #0 + Ascii_offset
0D61 85 77 STA High_hour
0D63 85 78 STA Low_hour
0D65 60 RTS
0D66
0D66 / DISPLAY TIME ON THE MODE7 SCREEN
0D66 / FIRST COLLECT START OF MODE7
0D66 / SCREEN FROM &350 AND &351.
0D66  / COPY THE ADDRESS TO A ZERO
0D66  / PAGE ADDRESS FOR POST-INDEX
0D66  / ADDRESSING. PUT INTO 'Y' AN
0D66  / OFFSET TO MOVE DISPLAY ACROSS
0D66  / TO RIGHT HAND SIDE OF SCREEN
0D66

0D66 AD 50 03  .DISPLAY LDA &350
0D69 85 70  STA &70
0D6B AD 51 03 LDA &351
0D6E 85 71  STA &71
0D70 A0 1C  LDY #&1C
0D72
0D72 A9 83  LDA #131 /YELLOW COLOUR
0D74 91 70  STA (&70),Y
0D76 C8  INY
0D77 A5 77  LDA High_hour
0D79 91 70  STA (&70),Y
0D7B A5 78  LDA Low_hour
0D7D C8  INY
0D7E 91 70  STA (&70),Y
0D80 A9 3A  LDA #58 / Ascii code for colon
0D82 C8  INY
0D83 91 70  STA (&70),Y
0D85 A5 75  LDA High_minute
0D87 C8  INY
0D88 91 70  STA (&70),Y
0D8A A5 76  LDA Low_minute
0D8C C8  INY
0D8D 91 70  STA (&70),Y
0D8F A9 3A  LDA #58 / Ascii code for colon
0D91 C8  INY
0D92 91 70  STA (&70),Y
0D94 A5 73  LDA High_second
0D96 C8  INY
0D97 91 70  STA (&70),Y
0D99 A5 74  LDA Low_second
0D9B C8  INY
0D9C 91 70  STA (&70),Y
0D9E C8  INY
0D9F A9 87  LDA #135 / WHITE COLOUR
0DA1 91 70  STA (&70),Y
0DA3 60  RTS
Function key 8 turns on the clock
Function key 9 turns off the clock

SCREEN REVOLVE

This program is really a clever party trick. With little effort it can be buried deep in your enemies' programs to cause them nightmares as their display seemed to wander round the screen. After you have assembled the program you must wait for some address calculations. Then the screen will seem to go round and round. You cannot write to the screen in this program, but coupled with an interrupt, such as in 'Music while you list' or the 'MODE 7 Clock', it could prove devastating.

**Lines 10-110** The program will work in any MODE except MODE 7.

**Lines 120-200** We decided that sixty possible start points for the screen were sufficient. If you increase the number of screen points for a smoother display, then you will need to reduce the machine code delay routine. We have made it obvious that calculations are in radians.

**Lines 210-400** The actual program. We will be feeding locations &12 and &13 with 'new' starts of screen. The value held in these locations is 'HIMEM / 8'. The '+.5' in line 320 is the standard correction for the BASIC function INT.

**Lines 410 to end** You will need to read up about 'Sheila' in the *User Guide*. Jeremy Ruston's book is also useful. To access the 6845 video chip registers directly, you need to place the video chip register you want into &FE00. The operating system processes it. The value you want in any particular video chip register must go into &FE01. The video chip has registers just like the 6502 processor.

This program shows that there is more than one 'processor' in the BBC. In many micros the main processor has to handle all the VDU addressing. In the BBC Micro this is done by the separate 6845 video processor.
PROGRAM LISTING

10 REM *****************************************************************
20 REM *****  Written by Ian Clarke.  *****
30 REM *****  August 83  *****
40 REM *****************************************************************
50
60 REM -----------------------------------------------
70 REM define MODE, prepare m/code
80 REM and work out screen positions
90 REM ...............................  
100
110 MODE 4
120
130 DIM X%(60), Y%(60)
140 PROC Circle
150 FOR Circle__position% = 1 TO 60
160 LET Rad__value = Circle__position% / 60 * 2 * PI
170 LET X%(Circle__position%) = INT(((SIN(Rad__value) *20) +.5 )
180 LET Y%(Circle__position%) = INT(((COS(Rad__value) *20) +.5 )
190 NEXT Circle__position%
200
210 REM -----------------------------------------------
220 REM the main program. the start
230 REM of screen has the circle
240 REM offset added to it and the
250 REM final value is stored in two
260 REM registers ready for m/code.
270 REM ...............................  
280
290 REPEAT
300
310 FOR Screen__position% = 0 TO 60
320 LET Screen__start% = INT(((HIMEM/8) + X%(Screen__position%) + Y%(Screen__position%)*40)+.5)
330
340 LET ?&B00 = Screen__start% DIV 256
350 LET ?&B01 = Screen__start% MOD 256
360 CALL &B02
370 NEXT Screen_position%
380 UNTIL FALSE
390
400 END
410
420 REM ================================
430 REM start of machine code
440 REM ----------------------------
450
460 DEF PROCCircle
470
480 LET P% = &B02
490
500 [
510
520 /ACCESS REGISTER 12 OF THE 6845
530 /VIDEO CHIP THROUGH 'SHEILA'
540 /ADDRESS &FE00. USING &FE01 STORE
550 /THE CHOSEN VALUE IN THE 6845 CHIP
560
570      LDA #12
580      STA &FE00
590      LDA &B00
600      STA &FE01
610
620 /REPEAT THE PROCEDURE FOR REGISTER
630 /13 OF THE 6845 VIDEO CHIP
640
650      LDA #13
660      STA &FE00
670      LDA &B01
680      STA &FE01
690
700 /DEFINE LENGTH OF DELAY. TO REDUCE
710 /DELAY REDUCE THE VALUE IN 'X'
720
730      LDX #30
740      LDY #255
750 .DELAY% DEY
760      BNE DELAY%
770      DEX
780       BNE DELAY%
790       RTS
800       |
810
820 PRINT
830 PRINT "Please wait while addresses"
840 PRINT "are calculated for this effect"
850 PRINT
860 ENDPRepeat

>RUN
0B02
0B02
0B02  /ACCESS REGISTER 12 OF THE 6845
0B02  /VIDEO CHIP THROUGH 'SHEILA'
0B02  /ADDRESS &FE00. USING &FE01 STORE
0B02  /THE CHOSEN VALUE IN THE 6845 CHIP
0B02
0B02 A9 0C  LDA #12
0B04 8D 00  FE STA &FE00
0B07 AD 00  0B LDA &B00
0B0A 8D 01  FE STA &FE01
0B0D
0B0D  /REPEAT THE PROCEDURE FOR
0B0D  REGISTER
0B0D  /13 OF THE 6845 VIDEO CHIP
0B0D
0B0D A9 0D  LDA #13
0B0F 8D 00 FE  STA &FE00
0B12 AD 01 0B  LDA &B01
0B15 8D 01 FE  STA &FE01
0B18
0B18  /DEFINE LENGTH OF DELAY. TO
0B18  REDUCE
0B18  /DELAY REDUCE THE VALUE IN 'X'
0B18
0B18 A2 1E  LDX #30
0B1A A0 FF  LDY #255
0B1C 88  .DELAY% DEY
0B1D D0 FD  BNE DELAY%
0B1F CA  DEX
0B20 D0 FA     BNE DELAY%
0B22 60        RTS

Please wait while addresses
are calculated for this effect

A SIMPLE GAME

To be fun, all good games are simple. I will leave it to you to
embellish this one, but you can play it happily as it stands.

You will see on the screen a ‘Jedi’ style rod of light. This rod of
light grows at the rate set at the beginning of the game -
depending on skill level and speed. You use the ‘A’, ‘Z’, ‘<’ and
’>’ keys to change the direction of ‘growth’ of the rod. Once the
rod reaches the edge of the screen you have lost and the time
taken is recorded.

The ‘A’ key controls ‘up’
The ‘Z’ key controls ‘down’
The ‘<’ key controls ‘left’
The ‘>’ key controls ‘right’

A good speed is ‘40’ and a good skill level is ‘30’.

For improvements, I suggest a yellow border round the edge
of the screen and a red rod. A score card is needed - and
perhaps a bit of sound. All this should be within your capabilities.

Lines 10-330 The memory locations for the program are
reserved. The ‘XLOW’, etc, from lines 210-240 are the coordinates
of the end of the rod. The ‘left’ directions, etc (lines 260-290),
hold the offset in each direction which will be added on during
the game for movements in that particular direction.
Lines 340-600 Initial values are set for the speed and rod
movement into the top left corner.

Lines 610-790 The user’s requirements are entered and
validated. You can add program instructions at this stage if you
wish.

Lines 800-1230 These routines simply draw a line for the rod
after a key ‘scan’.

Lines 1240-1460 The input buffer is cleared at 1260 to 1280.
Using OSBYTE 129 the keyboard is tested and one of the four
following keys is trapped.

**Lines 1470-1670** The new size of the rod is worked out in following sub-routines. Meanwhile an on-screen check is made. The delay loop is simply a ‘nested’ loop. Changing values here does not change anything except the speed with which the rod is re-displayed.

**Lines 1800-2570** The principle here is that when you press a key to change direction, the previous (different) direction is ‘zeroed’ and the new direction is ‘incremented’. If it so happens that you are already travelling in that direction, this is added again to make a new speed in that direction. Your errors are compounded - so to speak.

You will notice at line 1650 that if you are not inside the screen you return to BASIC.

**Lines 2680-2760** These are self-evident.

---

**PROGRAM LISTING**

```
10 REM ***********************************************
20 REM * MOVING GRAPHICS PROGRAM *
30 REM * BY ANDREW PUSEY. *
40 REM * AUGUST 1983. *
50 REM ***********************************************
60
70 REM ------------------------------------------
80 REM allocate machine code and
90 REM data memory locations
100 REM .........................
110
120 MODE 7
130 DIM Space 500
140 LET START = Space + 16
145 LET Data = Space
150
160 REM ------------------------------------------
170 REM give variable names to the
180 REM data locations
190 REM .........................
```
200
210 LET XLOW = Data + 0
220 LET XHIGH = Data + 1
230 LET YLOW = Data + 2
240 LET YHIGH = Data + 3
250 LET RDOU T = Data + 4
260 LET Left = Data + 5
270 LET Right = Data + 6
280 LET Up = Data + 7
290 LET Down = Data + 8
300 LET SPEED = Data + 9
310 LET SPEED2 = Data + 10
320
330 REM ===============================
340 REM set up machine operating
350 REM system vectors
360 REM .........................
370
380 LET RDOU T LOW = XLOW MOD 256
390 LET RDOU T HIGH = XLOW DIV 256
400 LET OSWRCH = &FFEE
410 LET OSBYTE = &FFF4
420 LET OSWORD = &FFF1
430
440 REM ===============================
450 REM allocate values to the data
460 REM memory locations
470 REM .........................
480
490 LET ?SPEED = 15: REM initial speed
500 LET ?SPEED2 = 0
510 LET ?XLOW = 130
520 LET ?XHIGH = 2
530 LET ?YLOW = 10
540 LET ?YHIGH = 2
550 LET ?Left = 10
560 LET ?Right = 0
570 LET ?Up = 10
580 LET ?Down = 0
590
600 REM ===============================
610 REM get in skill level. The
620 REM easiest is level 255
630 REM also get in speed. Fastest
640 REM is speed 60
650 REM ............................
660
670 REM
680 CLS
690 PRINT CHR$(133);
700 INPUT "SKILL LEVEL (0-255)",RATE
710 UNTIL RATE >= 0 AND RATE <= 255
720 REM
730 CLS
740 PRINT CHR$(129);
750 INPUT "ROD SPEED (10-60)",Speed
760 UNTIL Speed >= 10 AND Speed <= 60
770 LET ?SPEED = Speed
780
790 REM ================
800 REM start of machine code
810 REM ........................
820
830 FOR PASS = 0 TO 3 STEP 3
840
850 P%=START
860 []
870 OPT PASS
880 JMP Key
890
900 / ROUTINE TO MOVE END OF STICK TO
910 / MIDDLE OF SCREEN AND DRAW LINE
920 / TO OTHER END OF STICK.
930
940.Move LDA #25
950 JSR OSWRCH
960 LDA #04
970 JSR OSWRCH
980 LDA #138
990 JSR OSWRCH
1000 LDA #02
1010 JSR OSWRCH
1020 LDA #00
1030 JSR OSWRCH

161
<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1040</td>
<td>LDA #02</td>
</tr>
<tr>
<td>1050</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1040</td>
<td>LDA #02</td>
</tr>
<tr>
<td>1050</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1060</td>
<td></td>
</tr>
<tr>
<td>1070</td>
<td>/ NOW DRAW TO END OF THE STICK</td>
</tr>
<tr>
<td>1080</td>
<td></td>
</tr>
<tr>
<td>1090</td>
<td>LDA #25</td>
</tr>
<tr>
<td>1100</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1110</td>
<td>LDA #06</td>
</tr>
<tr>
<td>1120</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1130</td>
<td>LDA XLOW</td>
</tr>
<tr>
<td>1140</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1150</td>
<td>LDA XHIGH</td>
</tr>
<tr>
<td>1160</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1170</td>
<td>LDA YLOW</td>
</tr>
<tr>
<td>1180</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1190</td>
<td>LDA YHIGH</td>
</tr>
<tr>
<td>1200</td>
<td>JSR OSWRCH</td>
</tr>
<tr>
<td>1210</td>
<td>RTS</td>
</tr>
<tr>
<td>1220</td>
<td></td>
</tr>
<tr>
<td>1230</td>
<td></td>
</tr>
<tr>
<td>1240</td>
<td>/ SCAN KEYBOARD FOR KEY PRESSED</td>
</tr>
<tr>
<td>1250</td>
<td></td>
</tr>
<tr>
<td>1260.Key</td>
<td>LDA #21</td>
</tr>
<tr>
<td>1270</td>
<td>LDX #1</td>
</tr>
<tr>
<td>1280</td>
<td>JSR OSBYTE</td>
</tr>
<tr>
<td>1290</td>
<td>DEX</td>
</tr>
<tr>
<td>1300</td>
<td>LDA #129</td>
</tr>
<tr>
<td>1310</td>
<td>LDY #00</td>
</tr>
<tr>
<td>1320</td>
<td>JSR OSBYTE</td>
</tr>
<tr>
<td>1330</td>
<td>TXA</td>
</tr>
<tr>
<td>1340</td>
<td></td>
</tr>
<tr>
<td>1350</td>
<td>/ CHECK FOR STICK CONTROL KEYS</td>
</tr>
<tr>
<td>1360</td>
<td></td>
</tr>
<tr>
<td>1370</td>
<td>CMP #ASC(&quot;A&quot;)</td>
</tr>
<tr>
<td>1380</td>
<td>BEQ up</td>
</tr>
<tr>
<td>1390</td>
<td>CMP #ASC(&quot;Z&quot;)</td>
</tr>
<tr>
<td>1400</td>
<td>BEQ down</td>
</tr>
<tr>
<td>1410</td>
<td>CMP #ASC(&quot;&quot;,&quot;&quot;)</td>
</tr>
<tr>
<td>1420</td>
<td>BEQ left</td>
</tr>
<tr>
<td>1430</td>
<td>CMP #ASC(&quot;.&quot;,&quot;&quot;)</td>
</tr>
</tbody>
</table>
1440    BEQ right
1450
1460 / GO AND CALCULATE NEW POSITION
1470
1480.JUMP5    JSR add
1490
1500 / GO AND DRAW STICK ON SCREEN
1510
1520        JSR Move
1530
1540 / CHECK TO SEE IF END OF STICK
1550 / IS OUTSIDE LIMITS OF SCREEN
1560 / BY CHECKING THE PIXEL COLOUR
1570
1580       LDX #RDOTLOW
1590       LDY #RDOTHIGH
1600       LDA #09
1610       JSR OSWORD
1620       LDA RDOT
1630       CMP #255
1640       BNE INSIDE
1650       RTS
1660
1670 / ROUTINE TO WAIT FOR A WHILE
1680
1690.INSIDE LDY#55
1700.JUMP6  DEY
1710       LDX#255
1720.JUMP7  DEX
1730       TXA
1740       BNE JUMP7
1750       TYA
1760       BNE JUMP6
1770
1780 / PLOT STICK AGAIN TO REMOVE IT
1790
1800       JSR Move
1810       JMP Key
1820
1830 / SET DIRECTION TO UP
1840
1850.up    LDA SPEED
1860 STA Up
1870 LDA #00
1880 STA Down
1890 JMP JUMP5
1900
1910 / SET DIRECTION TO DOWN
1920
1930.down LDA SPEED
1940 STA Down
1950 LDA #00
1960 STA Up
1970 JMP JUMP5
1980
1990 / SET DIRECTION TO LEFT
2000
2010.left LDA SPEED
2020 STA Left
2030 LDA #00
2040 STA Right
2050 JMP JUMP5
2060
2070 / SET DIRECTION TO RIGHT
2080
2090.right LDA SPEED
2100 STA Right
2110 LDA #00
2120 STA Left
2130 JMP JUMP5
2140
2150 / ROUTINE TO CALCULATE NEW
2160 / POSITION OF STICK
2170
2180.add CLC
2190 INC SPEED2
2200 LDA SPEED2
2210 CMP #RATE
2220 BEQ add2
2230.add3 LDA YLOW
2240 ADC Up
2250 STA YLOW
2260 BCS JUMP1
2270 SEC
<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2280</td>
<td>LDA YLOW</td>
</tr>
<tr>
<td>2290</td>
<td>SBC Down</td>
</tr>
<tr>
<td>2300</td>
<td>STA YLOW</td>
</tr>
<tr>
<td>2310</td>
<td>BCC JUMP2</td>
</tr>
<tr>
<td>2320</td>
<td>CLC</td>
</tr>
<tr>
<td>2330</td>
<td>LDA XLOW</td>
</tr>
<tr>
<td>2340</td>
<td>ADC Right</td>
</tr>
<tr>
<td>2350</td>
<td>STA XLOW</td>
</tr>
<tr>
<td>2360</td>
<td>BCS JUMP3</td>
</tr>
<tr>
<td>2370</td>
<td>SEC</td>
</tr>
<tr>
<td>2380</td>
<td>LDA XLOW</td>
</tr>
<tr>
<td>2390</td>
<td>SBC Left</td>
</tr>
<tr>
<td>2400</td>
<td>STA XLOW</td>
</tr>
<tr>
<td>2410</td>
<td>BCC JUMP4</td>
</tr>
<tr>
<td>2420</td>
<td>RTS</td>
</tr>
<tr>
<td>2430</td>
<td>LDA #00</td>
</tr>
<tr>
<td>2440</td>
<td>STA SPEED2</td>
</tr>
<tr>
<td>2450</td>
<td>INC SPEED</td>
</tr>
<tr>
<td>2460</td>
<td>JMP add3</td>
</tr>
<tr>
<td>2470</td>
<td></td>
</tr>
<tr>
<td>2480</td>
<td>JUMP1 INC YHIGH</td>
</tr>
<tr>
<td>2490</td>
<td>RTS</td>
</tr>
<tr>
<td>2500</td>
<td>JUMP2 DEC YHIGH</td>
</tr>
<tr>
<td>2510</td>
<td>RTS</td>
</tr>
<tr>
<td>2520</td>
<td>JUMP3 INC XHIGH</td>
</tr>
<tr>
<td>2530</td>
<td>RTS</td>
</tr>
<tr>
<td>2540</td>
<td>JUMP4 DEC XHIGH</td>
</tr>
<tr>
<td>2550</td>
<td>RTS</td>
</tr>
<tr>
<td>2560</td>
<td>]</td>
</tr>
<tr>
<td>2570</td>
<td>NEXTPASS</td>
</tr>
<tr>
<td>2580</td>
<td></td>
</tr>
<tr>
<td>2590</td>
<td>REM ...............</td>
</tr>
<tr>
<td>2600</td>
<td>REM end of machine code</td>
</tr>
<tr>
<td>2610</td>
<td>REM ===============</td>
</tr>
<tr>
<td>2620</td>
<td></td>
</tr>
<tr>
<td>2630</td>
<td>REM ===============</td>
</tr>
<tr>
<td>2640</td>
<td>REM main game</td>
</tr>
<tr>
<td>2650</td>
<td>REM ...............</td>
</tr>
<tr>
<td>2660</td>
<td></td>
</tr>
<tr>
<td>2670</td>
<td>MODE5</td>
</tr>
<tr>
<td>2680</td>
<td>TIME = 0</td>
</tr>
<tr>
<td>2690</td>
<td>CALL START</td>
</tr>
</tbody>
</table>
2700 MODE7
2710 PRINT"CHR$(130)""YOU LASTED ";TIME DIV 100;
   "SECONDS"
2720 PRINT"CHR$(130)"" AT SKILL LEVEL ";RATE
2730 PRINT"CHR$(130)"" AT SPEED ";Speed
2740 FORT=1TO2000:A$=INKEY$(0):NEXT
2750 END

RUN

SKILL LEVEL (0-255)?40
ROD SPEED (10-60)?50
2ED3
2ED3
2ED3  OPT PASS
2ED3 4C 17 2F JMP Key
2ED6
2ED6  / ROUTINE TO MOVE END OF STICK TO
2ED6  / MIDDLE OF SCREEN AND DRAW LINE
2ED6  / TO OTHER END OF STICK.
2ED6
2ED6 A9 19  .Move  LDA #25
2ED8 20 EE FF  JSR OSWRCH
2EDB A9 04  LDA #04
2EDD 20 EE FF  JSR OSWRCH
2EE0 A9 8A  LDA #138
2EE2 20 EE FF  JSR OSWRCH
2EE5 A9 02  LDA #02
2EE7 20 EE FF  JSR OSWRCH
2EEA A9 00  LDA #00
2EEC 20 EE FF  JSR OSWRCH
2EFD A9 02  LDA #02
2EF1 20 EE FF  JSR OSWRCH
2EF4
2EF4  / NOW DRAW TO END OF THE STICK
2EF4
2EF4 A9 19  LDA #25
2EF6 20 EE FF  JSR OSWRCH
2EF9 A9 06  LDA #06
2EFB 20 EE FF  JSR OSWRCH
2EFE 2D CF 2E  LDA XLOW
2F01 20 EE FF  JSR OSWRCH
2F04 AD C4 2E LDA XHIGH
2F07 20 EE FF JSR OSWRCH
2F0A AD C5 2E LDA YLOW
2F0D 20 EE FF JSR OSWRCH
2F10 AD C6 2E LDA YHIGH
2F13 20 EE FF JSR OSWRCH
2F16 60 RTS
2F17
2F17 / SCAN KEYBOARD FOR KEY PRESSED
2F17
2F17 A9 15 .Key LDA #21
2F19 A2 01 LDX #1
2F1B 20 F4 FF JSR OSBYTE
2F1E CA DEX
2F1F A9 81 LDA #129
2F21 A0 00 LDY #00
2F23 20 F4 FF JSR OSBYTE
2F26 8A TXA
2F27
2F27 / CHECK FOR STICK CONTROL KEYS
2F27
2F27 C9 41 CMP #ASC("
    
A")
2F29 F0 35 BEQ up
2F2B C9 5A CMP #ASC("
    
Z")
2F2D F0 3F BEQ down
2F2F C9 2C CMP #ASC("
    
,"
    
")
2F31 F0 49 BEQ left
2F33 C9 2E CMP #ASC("
    
."
    
")
2F35 F0 53 BEQ right
2F37
2F37 / GO AND CALCULATE NEW POSITION
2F37
2F37 20 98 2F JUMP5 JSR add
2F3A
2F3A / GO AND DRAW STICK ON SCREEN
2F3A
2F3A 20 D6 2E JSR Move
2F3D
2F3D / CHECK TO SEE IF END OF STICK
2F3D / IS OUTSIDE LIMITS OF SCREEN
2F3D / BY CHECKING THE PIXEL COLOUR
2F3D
2F3D A2 C3  LDX #RDOUTLOW
2F3F A0 2E  LDY #RDOUTHIG
2F41 A9 09  LDA #09
2F43 20 F1 FF  JSR OSWORD
2F46 AD C7 2E  LDA RDOT
2F49 C9 FF  CMP #255
2F4B D0 01  BNE INSIDE
2F4D 60  RTS
2F4E
2F4E / ROUTINE TO WAIT FOR A WHILE
2F4E
2F4E A0 37  .INSIDE LDY#55
2F50 88  .JUMP6 DEY
2F51 A2 FF  LDX#255
2F53 CA  .JUMP7 DEX
2F54 8A  TXA
2F55 D0 FC  BNE JUMP7
2F57 98  TYA
2F58 D0 F6  BNE JUMP6
2F5A
2F5A / PLOT STICK AGAIN TO REMOVE IT
2F5A
2F5A 20 D6 2E  JSR Move
2F5D 4C 17 2F  JMP Key
2F60
2F60 / SET DIRECTION TO UP
2F60
2F60 AD CC 2E .up  LDA SPEED
2F63 8D CA 2E STA Up
2F66 A9 00  LDA #00
2F68 8D CB 2E STA Down
2F6B 4C 37 2F  JMP JUMP5
2F6E
2F6E / SET DIRECTION TO DOWN
2F6E
2F6E AD CC 2E .down  LDA SPEED
2F71 8D CB 2E STA Down
2F74 A9 00  LDA #00
2F76 8D CA 2E STA Up
2F79 4C 37 2F  JMP JUMP5
2F7C

168
2F7C          / SET DIRECTION TO LEFT
2F7C
2F7C AD CC 2E .left  LDA SPEED
2F7F 8D C8 2E STA Left
2F82 A9 00  LDA #00
2F84 8D C9 2E STA Right
2F87 4C 37 2F JMP JUMP5
2F8A
2F8A          / SET DIRECTION TO RIGHT
2F8A
2F8A AD CC 2E .right LDA SPEED
2F8D 8D C9 2E STA Right
2F90 A9 00  LDA #00
2F92 8D C8 2E STA Left
2F95 4C 37 2F JMP JUMP5
2F98
2F98          / ROUTINE TO CALCULATE NEW
2F98          / POSITION OF STICK
2F98
2F98 18    .add  CLC
2F99 EE CD 2E INC SPEED2
2F9C AD CD 2E LDA SPEED2
2F9F C9 28  CMP #RATE
2FA1 F0 30  BEQ add2
2FA3 AD C5 2E .add3  LDA YLOW
2FA6 6D CA 2E ADC Up
2FA9 8D C5 2E STA YLOW
2FAC B0 30  BCS JUMP1
2FAE 38  SEC
2FAF AD C5 2E LDA YLOW
2FB2 ED CB 2E SBC Down
2FB5 8D C5 2E STA YLOW
2FB8 90 28  BCC JUMP2
2FBA 18  CLC
2FBB AD C3 2E LDA XLOW
2FBE 6D C9 2E ADC Right
2FC1 8D C3 2E STA XLOW
2FC4 B0 20  BCS JUMP3
2FC6 38  SEC
2FC7 AD C3 2E LDA XLOW
2FCA ED C8 2E SBC Left
2FCD 8D C3 2E STA XLOW
LADDER DRAWING

This program again illustrates machine code graphics. It simply draws ladders onto the screen at any specified points. The ladders can, within reason, be any height and any width. The author claims inspiration from 'Killer Gorilla' type ladders. As long as you are not 'filling' the screen with colour, the machine code can be quite fast. These ladders seem to be instantaneous. Using other techniques you have learnt, you will be able to colour them. The program will not work in MODE 7.

Lines 10-160 The first seven locations for machine code are in fact allocated to program data, hence line 150.

Lines 170-360 The ladders are better in MODE 1, but suit yourself.

Lines 350-450 The 'X' coordinate needs two memory locations to be stored as it can range from 0 to approximately 1000. This is
a number larger than can be held in one memory location. This data will be held 'on the stack'.

**Lines 460-580** The start data on the initial 'X' and 'Y' coordinates is accessed and the graphics cursor is moved ready for the first vertical strut. Remember that VDU 25,4, etc., is the BASIC command MOVE.

**Lines 590-720** The line is drawn to the top of the ladder.

**Lines 720 & 730** This is the count for the two vertical struts of the ladder.

**Lines 750-840** The separation of the vertical struts is added on to the 'X' axis.

**Lines 850-980** The vertical distance between each rung is fixed at line 910 and this is added to the 'Y' coordinate to find where the rungs should start.

**Lines 990-1470** The ladder rungs are now filled into the ladder. Data stores 2 and 3 are used to hold the current 'Y' coordinate being drawn and this is checked (lines 1370-1450) with the data given at the start of the program.

**Lines 1480-end** A little keyboard entry in BASIC with data validation. The MODs and the DIVs are needed to split the 'X' and 'Y' coordinates between two memory stores as they can range outside the 255 maximum value that can be represented in one memory location.

---

**PROGRAM LISTING**

10 REM ******************************************************
20 REM ** Written by Ian Clarke. **
30 REM ** JULY 83 **
40 REM ******************************************************
50
60 REM =----------------------------------
70 REM establish program variables
80 REM with data stored before the
90 REM machine code
100 REM .........................
110
120 DIM Space% 255
130 LET OSWRCH = &FFEE
140 LET Data% = Space%
150 Start_of_program% = Space% + 7
160
170 REM -----------------------------------
180 REM start of machine code program
190 REM .........................
200
210 FOR Pass% = 0 TO 3 STEP 3
220
230 LET P% = Start_of_program%
240
250 [ ]
260 OPT Pass%
270
280 / SET UP MODE 1 SCREEN
290
300 LDA #22
310 JSR OSWRCH
320 LDA #1
330 JSR OSWRCH
340
350 /SET UP THE COUNTER
360
370 .No_mode_change% LDX #0
380
390 /PUSH THE 'X' POSITION ONTO STACK
400
410 LDA Data%
420 PHA
430 LDA Data%+1
440 PHA
450
460 /MOVE TO START POINTS
470
480 .Main__program% LDA #25
490 JSR OSWRCH
500 LDA #4
510    JSR OSWRCH
520    LDY #0
530   .Move__to__point%   LDA Data%,Y
540    JSR OSWRCH
550    INY
560    CPY #4
570    BNE Move__to__point%
580
590   /DRAW TO FOLLOWING POINTS
600
610     LDA #25
620    JSR OSWRCH
630     LDA #5
640    JSR OSWRCH
650     LDA Data%
660    JSR OSWRCH
670     LDA Data% + 1
680    JSR OSWRCH
690     LDA Data% + 4
700    JSR OSWRCH
710     LDA Data% + 5
720    JSR OSWRCH
730     INX
740     CPX #2
750    BEQ Ladder__top%
760     LDA Data%
770     CLC
780     ADC Data% + 6
790     STA Data%
800    BCC Main__program%
810     LDY Data% + 1
820     INY
830     STY Data% + 1
840    JMP Main__program%
850   .Ladder__top%     PLA
860     TAY
870     PLA
880     TAX
890     LDA Data%+2
900     CLC
910     ADC #20
920     STA Data%+2
930          BCC Ladder__rungs%
940          LDA Data%+3
950          CLC
960          ADC #1
970          STA Data%+3
980
990          /MOVE TO FOLLOWING POINTS
1000
1010         .Ladder__rungs%    LDA #25
1020         JSR OSWRCH
1030         LDA #4
1040         JSR OSWRCH
1050         LDA Data%
1060         JSR OSWRCH
1070         LDA Data%+1
1080         JSR OSWRCH
1090         LDA Data%+2
1100         JSR OSWRCH
1110         LDA Data%+3
1120         JSR OSWRCH
1130
1140         /MOVE TO FOLLOWING POINTS
1150
1160         LDA #25
1170         JSR OSWRCH
1180         LDA #5
1190         JSR OSWRCH
1200         TXA
1210         JSR OSWRCH
1220         TYA
1230         JSR OSWRCH
1240         LDA Data% + 2
1250         JSR OSWRCH
1260         LDA Data% + 3
1270         JSR OSWRCH
1280         CLC
1290         LDA Data% + 2
1300         ADC #37
1310         STA Data% + 2
1320         BCC End__check1%
1330         LDA Data% + 3
1340         CLC
1350    ADC #1
1360    STA Data% + 3
1370 .End_check1%     LDA Data% + 5
1380    CMP Data% + 3
1390    BEQ End_check2%
1400    BMI End_of_program%
1410    JMP Ladder_rungs%
1420 .End_check2%     LDA Data%+4
1430    CMP Data%+2
1440    BCC End_of_program%
1450    JMP Ladder_rungs%
1460 .End_of_program%  RTS
1470
1480 |
1490
1500 NEXT Pass%
1510
1520 REM END OF MACHINE CODE
1530 REM .........................
1540
1550 INPUT"Press the RETURN key when ready"A$
1560 MODE 4
1570 REPEAT
1580 VDU 28,0,31,39,30
1590 INPUT "Enter the start X "Start_of_X_axis% 
1600 IF Start_of_X_axis% < 0 OR Start_of_X_axis% > 1000 GOTO 1590
1610 INPUT "Enter the start Y "Start_of_Y_axis%
1620 IF Start_of_Y_axis% < 0 OR Start_of_Y_axis% > 1000 GOTO 1610
1630 INPUT "Enter the finish Y "Finish_of_Y_axis%
1640 IF Finish_of_Y_axis% < 0 OR Finish_of_Y_axis% > 1000 GOTO 1630
1650 IF Start_of_Y_axis% >= Finish_of_Y_axis% THEN
1660    CLS:UNTIL FALSE
1660 INPUT "Enter the gap size "Gap_size%
1670 IF Gap_size% < 1 OR Gap_size% > 255 THEN
1670    CLS:UNTIL FALSE
1680 INPUT "Do you want a change of mode (Y or N) "YN$
1690 IF LEFT$(YN$,1)<>"N" AND LEFT$(YN$,1)<>"Y" THEN UNTIL FALSE
1700
1710 REM ---------------------------------------
1720 REM set up the data
1730 REM ....................
1740
1750 LET ?Data% = Start_of_X_axis% MOD 256
1760 LET Data%?1 = Start_of_X_axis% DIV 256
1770 LET Data%?2 = Start_of_Y_axis% MOD 256
1780 LET Data%?3 = Start_of_Y_axis% DIV 256
1790 LET Data%?4 = Finish_of_Y_axis% MOD 256
1800 LET Data%?5 = Finish_of_Y_axis% DIV 256
1810 LET Data%?6 = Gap_size%
1820
1830 IF LEFT$(YN$,1)= "Y" THEN CALL Start_of_
    program% ELSE CALL No_mode_change%
1840 UNTIL FALSE

>RUN
2D80
2D80
2D80
2D80
2D80
2D80
/ SET UP MODE 1 SCREEN
2D80
2D80 A9 16  LDA #22
2D82 20 EE FF  JSR OSWRCH
2D85 A9 01  LDA #1
2D87 20 EE FF  JSR OSWRCH
2D8A
2D8A  /SET UP THE COUNTER
2D8A
2D8A A2 00  .No_mode_change% LDX #0
2D8C
2D8C  /PUSH THE 'X' POSITION ONTO STACK
2D8C
2D8C AD 79 2D  LDA Data%
2D8F 48  PHA
2D90 AD 7A 2D  LDA Data%+1
2D93 48  PHA
2D94
2D94  /MOVE TO START POINTS
2D94
2D94 A9 19 .Main_program% LDA #25
2D96 20 EE FF JSR OSWRCH
2D99 A9 04 LDA #4
2D9B 20 EE FF JSR OSWRCH
2D9E A0 00 LDY #0
2DA0 B9 79 2D .Move_to_point% LDA Data%,Y
2DA3 20 EE FF JSR OSWRCH
2DA6 C8 INY
2DA7 C0 04 CPY #4
2DA9 D0 F5 BNE Move_to_point%
2DAB
2DAB /DRAW TO FOLLOWING POINTS
2DAB
2DAB A9 19 LDA #25
2DAD 20 EE FF JSR OSWRCH
2DB0 A9 05 LDA #5
2DB2 20 EE FF JSR OSWRCH
2DB5 AD 79 2D LDA Data%
2DB8 20 EE FF JSR OSWRCH
2DBB AD 7A 2D LDA Data% + 1
2DBE 20 EE FF JSR OSWRCH
2DC1 AD 7D 2D LDA Data% + 4
2DC4 20 EE FF JSR OSWRCH
2DC7 AD 7E 2D LDA Data% + 5
2DCA 20 EE FF JSR OSWRCH
2DCE E8 INX
2DCE E0 02 CPX #2
2DD0 F0 16 BEQ Ladder_top%
2DD2 AD 79 2D LDA Data%
2DD5 18 CLC
2DD6 6D 7F 2D ADC Data% + 6
2DD9 8D 79 2D STA Data%
2DDC 90 B6 BCC Main_program%
2DDE AC 7A 2D LDA #1
2DE1 C8 INY
2DE2 8C 7A 2D STY Data% + 1
2DE5 4C 94 2D JMP Main_program%
2DE8 68 .Ladder_top% PLA
2DE9 A8 TAY
2DEA 68 PLA
2DEB AA TAX
2DEC AD 7B 2DLDA Data%+2
2DEF 18   CLC
2DF0 69 14  ADC #20
2DF2 8D 7B 2D STA Data%+2
2DF5 90 09  BCC Ladder__rungs%
2DF7 AD 7C 2D LDA Data%+3
2DFA 18   CLC
2DFB 69 01  ADC #1
2DFD 8D 7C 2D STA Data%+3
2E00
2E00     /MOVE TO FOLLOWING POINTS
2E00
2E00 A9 19  .Ladder__rungs%   LDA #25
2E02 20 EE FF JSR OSWRCH
2E05 A9 04  LDA #4
2E07 20 EE FF JSR OSWRCH
2E0A AD 79 2D LDA Data%
2E0D 20 EE FF JSR OSWRCH
2E10 AD 7A 2D LDA Data%+1
2E13 20 EE FF JSR OSWRCH
2E16 AD 7B 2D LDA Data%+2
2E19 20 EE FF JSR OSWRCH
2E1C AD 7C 2D LDA Data%+3
2E1F 20 EE FF JSR OSWRCH
2E22
2E22     /MOVE TO FOLLOWING POINTS
2E22
2E22 A9 19  LDA #25
2E24 20 EE FF JSR OSWRCH
2E27 A9 05  LDA #5
2E29 20 EE FF JSR OSWRCH
2E2C 8A   TXA
2E2D 20 EE FF JSR OSWRCH
2E30 98   TYA
2E31 20 EE FF JSR OSWRCH
2E34 AD 7B 2D LDA Data% + 2
2E37 20 EE FF JSR OSWRCH
2E3A AD 7C 2D LDA Data% + 3
2E3D 20 EE FF JSR OSWRCH
2E40 18   CLC
2E41 AD 7B 2D LDA Data% + 2
2E44 69 25  ADC #37
2E46 8D 7B 2D STA Data% + 2
2E49 90 09 BCC End__check1%
2E4B AD 7C 2D LDA Data% + 3
2E4E 18 CLC
2E4F 69 01 ADC #1
2E51 8D 7C 2D STA Data% + 3
2E54 AD 7E 2D .End__check1% LDA Data% + 5
2E57 CD 7C 2D CMP Data% + 3
2E5A F0 05 BEQ End__check2%
2E5C 30 0E BMI End__of__program%
2E5E 4C 00 2E JMP Ladder__rungs%
2E61 AD 7D 2D .End__check2% LDA Data%+4
2E64 CD 7B 2D CMP Data%+2
2E67 90 03 BCC End__of__program%
2E69 4C 00 2E JMP Ladder__rungs%
2E6C 60 .End__of__program% RTS
2E6D

Press the RETURN key when ready

Enter the start X 680
Enter the start Y 100
Enter the finish Y 900
Enter the gap size 50

Do you want change of mode (Y or N)N
FINALE

This book has missed a lot out - deliberately. It was always intended to get you started in assembly language, but to leave you to buy other texts to help you with detailed and advanced machine code. There are four books that I am happy to recommend:

*Assembly Language Programming for the BBC Microcomputer*
by Ian Birnbaum
Published by Macmillan
ISBN 0 333 34585 1
This is a detailed book with a mathematical bias. It has lots of good worked examples, and I have found it invaluable.

*Programming the 6502*
by Rodney Zaks
Published by Sybex
ISBN 0 89588 046 6
This is a detailed and useful book on the whole of the 6502 instruction set. There are some good general tips on addressing and data structures. In other Zaks books I have found the odd error, but I have not yet found one in this book. It is clearly laid out in large type.

*The BBC Micro: an expert guide*
by Mike James
Published by Granada
ISBN 0 246 12014 2
This is written in a friendly manner and reveals a lot about the inner workings of the BBC Micro. It does assume a fair degree of knowledge already, but after reading my book you should manage quite well.

*The BBC Micro Revealed*
by Jeremy Ruston
Published by Interface
ISBN 0 907563 15 5
This is for the ‘clever tricks’ brigade. Acorn do not guarantee that all that is revealed in this book will stay unchanged, as far as the operating system is concerned. Nevertheless it is a worthwhile purchase.
Finally, there is very little damage you can do to your micro by playing around with assembly language. The television or VDU monitor is more likely to suffer than the computer. You could by chance give instructions which may damage these. Be more careful if you have a disc system, as you could either 'wipe' discs or damage the head mechanism. With tape machines, you can relax. If you try to keep away from the clever tricks brigade when you start, then you are unlikely to run into problems.

You are going to see some extraordinary sights on your TV screen as some of the programs that you write reveal the awful 'bug'. Provided that you don't 'collapse' the TV display, don't worry about what you see. The BREAK key will not always clear all your problems unless you do a 'CONTROL BREAK' (v1.2 rom), repeated BREAK (v0.1 rom). If the worst comes to the worst, turn off the micro and start again. Always SAVE a copy of your program BEFORE you RUN it. A bug will almost certainly corrupt BASIC causing Bad Program errors.

Good luck. You'll need it!
APPENDICES

ASCII Table (BBC version) 1
Memory map assignments 2
Operating System Routines 3
6502 Instructions — Alphabetic 4
Hexadecimal Conversion Table 5
How Instructions Affect Flags 6
  Addressing Modes 7
Permitted Addressing Modes 8
# APPENDIX 1

## ASCII TABLE (BBC version)

CONTROL CODES 0-31 decimal, 00-1F hex.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>HEX</th>
<th>MODEs 0 to 6</th>
<th>MODE 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>no effect</td>
<td>no effect</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>next character to printer</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>start printing</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>stop printing</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>separate cursors</td>
<td>no effect</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>join cursors</td>
<td>no effect</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>turn on the VDU</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>sound the beeper</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>back the cursor</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>forward the cursor</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>down the cursor</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>up the cursor</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>clear text only</td>
<td>clear all screen</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>carriage return</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>stop continuous scroll</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>continuous scroll</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>clear graphics screen</td>
<td>no effect</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>define the text colour</td>
<td>no effect</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>define the graphics colour</td>
<td>no effect</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>define logical colours</td>
<td>no effect</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>default logical colours</td>
<td>no effect</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>turn off the VDU</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>select new mode</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>reprogram characters</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>no effect</td>
<td>no effect</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>start ‘plotting’</td>
<td>no effect</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>default text/graphics areas</td>
<td>no effect</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>define graphics area</td>
<td>no effect</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>define text area</td>
<td>no effect</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>define graphics origin</td>
<td>no effect</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>put cursor top left hand corner</td>
<td>(all MODEs)</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>place text cursor anywhere</td>
<td>(all MODEs)</td>
</tr>
</tbody>
</table>
## ASCII CODE

Normal codes 32-127 decimal, 20-7F hex

<table>
<thead>
<tr>
<th>Decimal</th>
<th>HEX</th>
<th>character</th>
<th>Decimal</th>
<th>HEX</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td>&lt;space&gt;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>!</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>#</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>$</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>%</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>‘apostrophe’</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>(</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>)</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>0</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>1</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>2</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>3</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>4</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>5</td>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>6</td>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>7</td>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>8</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>9</td>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>96</td>
<td>60</td>
<td>£</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>@</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>A</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>B</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>C</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>D</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>E</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>Decimal</td>
<td>HEX</td>
<td>character</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>6C</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>6D</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6E</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>6F</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>70</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>71</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>72</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>73</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>74</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>77</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>79</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>7A</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>7B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>7D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>7E</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>&lt;delete&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

APPENDIX 2

MEMORY MAP ASSIGNMENTS

FF00 – FFFF Operating System ROM
FE00 – FEFF Internal memory mapped input/output (SHEILA)
FD00 – FDFF External memory mapped input/output (JIM)
FC00 – FCFF External memory mapped input/output (FRED)
C000 – FBFF Operating System ROM
8000 – BFFF One or more languages ROMS (e.g. BASIC, PASCAL)
4000 – 7FFF Optional RAM on Model B
0000 – 3FFF always RAM
0E00 – Default setting of PAGE
0D80 – DFF allocated to machine operating system
0D00 – D7F Used by NMI routines (e.g. by Disc or Econet filing systems)
0C00 – CFF User defined character definitions
0B00 – BFF User defined function key definitions
0A00 – AFF RS423 receive, and cassette workspace
0900 – 9FF RS423 transmit, cassette, sound and speech workspace
0800 – 8FF Miscellaneous workspace
0400 – 7FF Language ROM workspace
0300 – 3FF Miscellaneous workspace
0200 – 2FF Operating system workspace and indirection vectors
0100 – 1FF 6502 stack
0000 – 0FF Zero page
### APPENDIX 3

**OPERATING SYSTEM ROUTINES**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Address</th>
<th>Vector Name</th>
<th>Address</th>
<th>Summary of function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UPTV</td>
<td>222</td>
<td>User print routine</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EVNTV</td>
<td>220</td>
<td>Event interrupt</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FSCV</td>
<td>21E</td>
<td>File system control entry</td>
</tr>
<tr>
<td>OSFIND</td>
<td>FFCE</td>
<td>FINDV</td>
<td>21C</td>
<td>Open or close a file</td>
</tr>
<tr>
<td>OSGBPB</td>
<td>FFD1</td>
<td>GBPBV</td>
<td>21A</td>
<td>Load or save a block of memory to file</td>
</tr>
<tr>
<td>OSBPUT</td>
<td>FFD4</td>
<td>BPUTV</td>
<td>218</td>
<td>Save a single byte to file from A</td>
</tr>
<tr>
<td>OSBGET</td>
<td>FFD7</td>
<td>BGETV</td>
<td>216</td>
<td>Load a single byte to A from file</td>
</tr>
<tr>
<td>OSARGS</td>
<td>FFDA</td>
<td>ARGSV</td>
<td>214</td>
<td>Load or save data about a file</td>
</tr>
<tr>
<td>OSFILE</td>
<td>FFDD</td>
<td>FILEV</td>
<td>212</td>
<td>Load or save a complete file</td>
</tr>
<tr>
<td>OSRDCH</td>
<td>FFE0</td>
<td>RDCHV</td>
<td>210</td>
<td>Read character (from keyboard) to A</td>
</tr>
<tr>
<td>OSASCI</td>
<td>FFE3</td>
<td>—</td>
<td>—</td>
<td>Write a character (to screen) from A plus LF if (A) = &amp;0D</td>
</tr>
<tr>
<td>OSNEWL</td>
<td>FFE7</td>
<td>—</td>
<td>—</td>
<td>Write LF, CR (&amp;oA, &amp;0D), to screen</td>
</tr>
<tr>
<td>OSWRCH</td>
<td>FFE9</td>
<td>WRCHV</td>
<td>20E</td>
<td>Write character (to screen) from A</td>
</tr>
<tr>
<td>OSWORD</td>
<td>FFF1</td>
<td>WORDV</td>
<td>20C</td>
<td>Perform miscellaneous OS operation using control block to pass parameters</td>
</tr>
<tr>
<td>OSBYTE</td>
<td>FFF4</td>
<td>BYTEV</td>
<td>20A</td>
<td>Perform miscellaneous OS operation using registers to pass parameters</td>
</tr>
<tr>
<td>OSCLI</td>
<td>FFF7</td>
<td>CLIV</td>
<td>208</td>
<td>Interpret the command line given</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>---------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Unrecognised IRQ vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>All IRQ vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Break vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reserved</td>
</tr>
</tbody>
</table>

**APPENDIX 4**

**6502 INSTRUCTIONS – ALPHABETIC**

| ADC   | Add with carry                     |
| AND   | Logical AND                        |
| ASL   | Arithmetic shift left              |
| BCC   | Branch if carry clear              |
| BCS   | Branch if carry set                |
| BEQ   | Branch if result = 0               |
| BIT   | Test bit                           |
| BMI   | Branch if minus                    |
| BNE   | Branch if not equal to 0           |
| BPL   | Branch if plus                     |
| BRK   | Break                              |
| BVC   | Branch if overflow clear           |
| BVS   | Branch if overflow set             |
| CLC   | Clear carry                        |
| CLD   | Clear decimal flag                 |
| CLI   | Clear interrupt disable            |
| CLV   | Clear overflow                     |
| CMP   | Compare to accumulator             |
| CPX   | Compare to X                       |
| CPY   | Compare to Y                       |
| DEC   | Decrement memory                   |
| DEX   | Decrement X                        |
| DEY   | Decrement Y                        |
| EOR   | Exclusive OR                       |
| INC   | Increment memory                   |
| INX   | Increment X                        |
| INY   | Increment Y                        |
| JMP   | Jump                               |
| JSR   | Jump to subroutines                |
| LDA   | Load accumulator                   |
| LDX   | Load X                             |
| LDY   | Load Y                             |
| LSR   | Logical shift right                |
| NOP   | No operation                       |
| ORA   | Logical OR                         |
| PHA   | Push A                             |
| PHP   | Push P status                      |
| PLA   | Pull A                             |
| PLP   | Pull P status                      |
| ROL   | Rotate left                        |
| ROR   | Rotate right                       |
| RTI   | Return from interrupt              |
| RTS   | Return from subroutine             |
| SBC   | Subtract with carry                |
| SEC   | Set carry                          |
| SED   | Set decimal                        |
| SEI   | Set interrupt disable              |
| STA   | Store accumulator                  |
| STX   | Store X                            |
| STY   | Store Y                            |
| TAX   | Transfer A to X                    |
| TAY   | Transfer A to Y                    |
| TSX   | Transfer SP to X                   |
| TXA   | Transfer X to A                    |
| TXS   | Transfer X to SP                   |
| TYA   | Transfer Y to A                    |
## APPENDIX 5

**HEXADECIMAL CONVERSION TABLE**

<table>
<thead>
<tr>
<th>HEX</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
</tr>
<tr>
<td>5</td>
<td>80</td>
<td>81</td>
<td>82</td>
<td>83</td>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
<td>88</td>
<td>89</td>
<td>90</td>
<td>91</td>
<td>92</td>
<td>93</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>6</td>
<td>96</td>
<td>97</td>
<td>98</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
<td>108</td>
<td>109</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>7</td>
<td>112</td>
<td>113</td>
<td>114</td>
<td>115</td>
<td>116</td>
<td>117</td>
<td>118</td>
<td>119</td>
<td>120</td>
<td>121</td>
<td>122</td>
<td>123</td>
<td>124</td>
<td>125</td>
<td>126</td>
<td>127</td>
</tr>
<tr>
<td>8</td>
<td>128</td>
<td>129</td>
<td>130</td>
<td>131</td>
<td>132</td>
<td>133</td>
<td>134</td>
<td>135</td>
<td>136</td>
<td>137</td>
<td>138</td>
<td>139</td>
<td>140</td>
<td>141</td>
<td>142</td>
<td>143</td>
</tr>
<tr>
<td>9</td>
<td>144</td>
<td>145</td>
<td>146</td>
<td>147</td>
<td>148</td>
<td>149</td>
<td>150</td>
<td>151</td>
<td>152</td>
<td>153</td>
<td>154</td>
<td>155</td>
<td>156</td>
<td>157</td>
<td>158</td>
<td>159</td>
</tr>
<tr>
<td>A</td>
<td>160</td>
<td>161</td>
<td>162</td>
<td>163</td>
<td>164</td>
<td>165</td>
<td>166</td>
<td>167</td>
<td>168</td>
<td>169</td>
<td>170</td>
<td>171</td>
<td>172</td>
<td>173</td>
<td>174</td>
<td>175</td>
</tr>
<tr>
<td>B</td>
<td>176</td>
<td>177</td>
<td>178</td>
<td>179</td>
<td>180</td>
<td>181</td>
<td>182</td>
<td>183</td>
<td>184</td>
<td>185</td>
<td>186</td>
<td>187</td>
<td>188</td>
<td>189</td>
<td>190</td>
<td>191</td>
</tr>
<tr>
<td>C</td>
<td>192</td>
<td>193</td>
<td>194</td>
<td>195</td>
<td>196</td>
<td>197</td>
<td>198</td>
<td>199</td>
<td>200</td>
<td>201</td>
<td>202</td>
<td>203</td>
<td>204</td>
<td>205</td>
<td>206</td>
<td>207</td>
</tr>
<tr>
<td>D</td>
<td>208</td>
<td>209</td>
<td>210</td>
<td>211</td>
<td>212</td>
<td>213</td>
<td>214</td>
<td>215</td>
<td>216</td>
<td>217</td>
<td>218</td>
<td>219</td>
<td>220</td>
<td>221</td>
<td>222</td>
<td>223</td>
</tr>
<tr>
<td>E</td>
<td>224</td>
<td>225</td>
<td>226</td>
<td>227</td>
<td>228</td>
<td>229</td>
<td>230</td>
<td>231</td>
<td>232</td>
<td>233</td>
<td>234</td>
<td>235</td>
<td>236</td>
<td>237</td>
<td>238</td>
<td>239</td>
</tr>
<tr>
<td>F</td>
<td>240</td>
<td>241</td>
<td>242</td>
<td>243</td>
<td>244</td>
<td>245</td>
<td>246</td>
<td>247</td>
<td>248</td>
<td>249</td>
<td>250</td>
<td>251</td>
<td>252</td>
<td>253</td>
<td>254</td>
<td>255</td>
</tr>
</tbody>
</table>
## APPENDIX 6

### HOW INSTRUCTIONS AFFECT FLAGS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Sign</th>
<th>Overflow</th>
<th>Break</th>
<th>Decimal</th>
<th>Interrupt</th>
<th>Zero</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>SBC</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>INC</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>INX</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>INY</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>DEX</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>DEY</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>LDX</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>LDY</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>STX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>STY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>TAX</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>TAY</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>TXA</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>TYA</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>TSX</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>TXS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BCS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BMI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BVC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>BVS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>

● means the FLAG MAY be changed as a result of the instruction.
## HOW INSTRUCTIONS AFFECT FLAGS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Sign</th>
<th>Overflow</th>
<th>Break</th>
<th>Decimal</th>
<th>Interrupt</th>
<th>Zero</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DIRECT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>CLV</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SEI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>STACK</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLP</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORA</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>COMPARE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPX</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPY</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SHIFT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASL</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>O R T H E R S</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>RTS</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* means the FLAG MAY be changed as a result of the instruction.
0 means a ZERO is placed in the flag.
1 means a ONE is placed in the flag.
APPENDIX 7

ADDRESSING MODES

The following instructions do NOT use any special addressing mode.

(machine code instruction shown in hex)

<table>
<thead>
<tr>
<th>BRK</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td>18</td>
</tr>
<tr>
<td>CLD</td>
<td>D8</td>
</tr>
<tr>
<td>CLI</td>
<td>58</td>
</tr>
<tr>
<td>CLV</td>
<td>88</td>
</tr>
<tr>
<td>DEX</td>
<td>CA</td>
</tr>
<tr>
<td>DEY</td>
<td>88</td>
</tr>
<tr>
<td>INX</td>
<td>E8</td>
</tr>
<tr>
<td>INY</td>
<td>C8</td>
</tr>
<tr>
<td>NOP</td>
<td>EA</td>
</tr>
<tr>
<td>PHA</td>
<td>48</td>
</tr>
<tr>
<td>PHP</td>
<td>08</td>
</tr>
<tr>
<td>PLA</td>
<td>68</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLP</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTI</td>
<td>40</td>
</tr>
<tr>
<td>RTS</td>
<td>60</td>
</tr>
<tr>
<td>SEC</td>
<td>38</td>
</tr>
<tr>
<td>SED</td>
<td>F8</td>
</tr>
<tr>
<td>SEI</td>
<td>78</td>
</tr>
<tr>
<td>TAX</td>
<td>AA</td>
</tr>
<tr>
<td>TAY</td>
<td>A8</td>
</tr>
<tr>
<td>TSX</td>
<td>BA</td>
</tr>
<tr>
<td>TXA</td>
<td>8A</td>
</tr>
<tr>
<td>TXS</td>
<td>9A</td>
</tr>
<tr>
<td>TYA</td>
<td>98</td>
</tr>
</tbody>
</table>
## APPENDIX 8

### PERMITTED ADDRESSING MODES

(machine code instruction shown in hex)

<table>
<thead>
<tr>
<th>LDA</th>
<th>LDX</th>
<th>LDY</th>
<th>STA</th>
<th>STX</th>
<th>STY</th>
<th>BCC</th>
<th>BCS</th>
<th>BEQ</th>
<th>BNE</th>
<th>BMI</th>
<th>BPL</th>
<th>BVC</th>
<th>BVS</th>
<th>ADC</th>
<th>SBC</th>
<th>INC</th>
<th>DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>AE</td>
<td>AC</td>
<td>8D</td>
<td>8E</td>
<td>8C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>A6</td>
<td>A4</td>
<td>85</td>
<td>86</td>
<td>84</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td>A2</td>
<td>A0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BD</td>
<td>BC</td>
<td>9D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9</td>
<td>BE</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>B4</td>
<td>95</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td></td>
<td>96</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>90</td>
<td>B0</td>
<td>F0</td>
<td>D0</td>
<td>30</td>
<td>10</td>
<td>50</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accumulator only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute</td>
</tr>
<tr>
<td>Zero page addressing</td>
</tr>
<tr>
<td>Immediate</td>
</tr>
<tr>
<td>Absolute indexed by 'x'</td>
</tr>
<tr>
<td>Absolute indexed by 'y'</td>
</tr>
<tr>
<td>Zero page indexed by 'x'</td>
</tr>
<tr>
<td>Zero page indexed by 'y'</td>
</tr>
<tr>
<td>Relative</td>
</tr>
<tr>
<td>Indirect</td>
</tr>
<tr>
<td>Indirect pre-index by 'x'</td>
</tr>
<tr>
<td>Indirect post-index by 'y'</td>
</tr>
</tbody>
</table>

### Moving Data Around | Branches | Maths
## PERMITTED ADDRESSING MODES
(machine code instruction shown in hex)

<table>
<thead>
<tr>
<th>AND</th>
<th>ORA</th>
<th>EOR</th>
<th>JMP</th>
<th>JSR</th>
<th>BIT</th>
<th>CMP</th>
<th>CPX</th>
<th>CPY</th>
<th>ASL</th>
<th>LSR</th>
<th>ROL</th>
<th>ROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>0D</td>
<td>4D</td>
<td>4C</td>
<td>20</td>
<td>2C</td>
<td>CD</td>
<td>EC</td>
<td>CC</td>
<td>0E</td>
<td>4E</td>
<td>2E</td>
<td>6E</td>
</tr>
<tr>
<td>25</td>
<td>05</td>
<td>45</td>
<td></td>
<td></td>
<td>24</td>
<td>C5</td>
<td>E4</td>
<td>C4</td>
<td>06</td>
<td>46</td>
<td>26</td>
<td>66</td>
</tr>
<tr>
<td>29</td>
<td>09</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td>C9</td>
<td>E0</td>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>1D</td>
<td>5D</td>
<td>DD</td>
<td></td>
<td>1E</td>
<td>5E</td>
<td>3E</td>
<td>7E</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>19</td>
<td>59</td>
<td>D9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>15</td>
<td>55</td>
<td>D5</td>
<td></td>
<td>16</td>
<td>56</td>
<td>36</td>
<td>76</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Addressing Modes
- **Accumulator only**
- **Absolute**
- **Zero page addressing**
- **Immediate**
- **Absolute indexed by 'x'**
- **Absolute indexed by 'y'**
- **Zero page indexed by 'x'**
- **Zero page indexed by 'y'**
- **Relative**
- **Indirect**
- **Indirect pre-index by 'y'**
- **Indirect post-index by 'x'**

### Logical, Jump, Compare, Shift etc.
Also available from Century Communications:

BOBBY CATCHES A BUG
Ray Hammond and Michael Cole

BOBBY MEETS A PIRATE
Ray Hammond and Michael Cole

COMPUTERS AND YOUR CHILD
Ray Hammond

DICTIONARY OF COMPUTING AND NEW INFORMATION TECHNOLOGY
A.J. Meadows, M. Gordon and A Singleton

THE ELECTRONIC MAIL HANDBOOK
Stephen Connell and A. Galbraith

MICROMAN
Gordon Pask and Susan Curran

MICROPHOBIA
Martin Honeysett

THE REALLY EASY GUIDE TO HOME COMPUTING: ZX SPECTRUM
Sue Beasley and Ruth Clarke

THE WAY THE NEW TECHNOLOGY WORKS
Ken Marsh

WHAT TO BUY FOR BUSINESS: A HANDBOOK OF NEW OFFICE TECHNOLOGY
John Derrick and Phillip Oppenheim

In association with 'Personal Computer World'

BEST OF PCW: SOFTWARE FOR THE BBC MICRO

BEST OF PCW: SOFTWARE FOR THE DRAGON

BEST OF PCW: SOFTWARE FOR THE SPECTRUM

THE CENTURY COMPUTER PROGRAMMING COURSE
Peter Morse and Ian Adamson

COMPUTER GAMESMANSHEP
David Levy

THE DATABASE PRIMER
Rose Deakin

35 EDUCATIONAL PROGRAMS FOR THE BBC MICRO
Ian Murray

EDUCATIONAL PROGRAMS FOR THE DRAGON 32
Ian Murray

EDUCATIONAL PROGRAMS FOR THE SPECTRUM
Ian Murray

INFORMATION TECHNOLOGY YEARBOOK
edited by Philip Hills

194
THE INTIMATE MACHINE
Neil Frude

THE MICROCOMPUTER HANDBOOK: A BUYER'S GUIDE
edited by Dick Olney

MICROCOMPUTING FOR BUSINESS: A USER'S GUIDE
edited by Dick Olney

MICROWARS ON THE COMMODORE 64
Humphrey Walwyn

THE ORIC HANDBOOK
Peter Lupton and Frazer Robinson

WRITING IN THE COMPUTER AGE
Andrew Fluegelman and Jeremy Hewes
INDEX
Compiled by Paul Nash

absolute addressing 63, 66, 108-9
accumulator register 10-11, 57
action fields 38, 39
ADC (ADd with Carry) 35, 97-100, 107, 109, 187, 189, 192
addition
accumulator 11
binary 4, 97-8
repeated 36-8
address coding 2
address storage 50, 64
ALGOL programming language 9
alphabet, coding of letters 3
ALU (arithmetic logic unit) 6, 7, 10, 57
AND instruction 113-14, 187
and (§) sign 46, 49, 53
approximate (~-) sign 50, 54, 122
ASCII codes 3, 46, 183
assembler, explained 8, 34-42
assembler directive 47, 49
assembly programs, defined 7-9
AUTO instruction 43
back slash sign (/) 82-3
back-to-front address 50, 64, 89-90
base address 64, 109
BASIC language 9
memory locations for 26, 185
BASIC programs
assembly language within 26, 43, 101
memory locations for 26, 27, 185
BCC (Branch if Carry flag is Clear) 107, 187
BCD (Binary Coded Decimal) 59
BCS (Branch if Carry flag is Set) 107, 187
binary arithmetic 3-5, 97-8
binary digits, defined 1
binary numbers, translation into decimal 49
hexadecimal 49, 55-6
BIT (test specific BIT) 117, 187
blank lines 121
blank spaces 38, 42
BMI (Branch if sign flag is Minus) 107, 187
BNE (Branch if Not Equal to zero) 65, 79, 81, 87, 109, 187
books to read 179-80
BPL (Branch if sign flag is Plus) 87, 107, 187
brackets [ 44; ( 113
branch instructions 66, 71, 75, 100, 107, 187, 189, 192
<break> key 15, 18, 32, 180
break flag 59
BRK (BReaK) 59, 117, 187
bugs, cause of 3, 87
bus 57
BVC (Branch if oVerflow flag Clear) 102, 107, 187
BVS (Branch if oVerflow flag set) 100, 107, 109, 187
CALL command 47, 122, 123
capital letters 35, 56
carry bit 69
carry flag 59, 69, 90, 103, 107
chaining programs 123
chips: see processors
CLC (CLear Carry flag) 97, 102, 107, 187
CLD (CLear Decimal flag) 92, 187
CLeaR flag instructions 92, 97, 102, 107, 117, 187, 190
CMP (COmpare) 91, 187
COBOL programming language 9
coding needs 2-3
commas 35, 42, 93
comments 36, 42, 82-3
CompAire instructions 11, 117, 187, 190, 193
conditional assembly 121-2
control codes 183-5
control unit 6, 7
CPU (central processing unit) 7, 8, 57
debugging of programs 3, 125
DEC (DECrement specific memory location 108, 187
decimal counting 52
decimal equivalents of
binary numbers 49, 50
hexadecimal numbers 54, 187
decimal flag 59, 92
DEFining commands 121-2
delimiters 35, 42
DIM statements 33-4, 64, 95
Disc Filing Systems (DFS), memory used by 15, 52
DIV commands 88-90, 95, 171
EBCDIC code 3
everse magnificent attention 17, 93-5
EOR (Exclusive OR) 115-16, 186
error listings 46, 48, 101
ters
byte 99, 101
keyboard reading 90
label 101
typing 42
<escape> key 15, 90-1
events, interrupt 16, 143
fields 35-6, 42, 46
flags affected by instructions 189-90
flags direct instructions 190
see Clear; SET
flags register 59
FOR...NEXT loops 42, 101
forward label references 41-2, 101
full stop starting labels 46
function keys 18
*FX calls 16, 17, 91-3

196
Assembly Programming Made Easy puts the magic world of machine code at your fingertips. Ian Murray has drawn on his teaching experience to present an easy beginner's guide to assembly language for the BBC Micro.

Using basic examples, cartoons and short and very simple programs you'll quickly learn how to put graphics on the screen, use the BBC operating system and bring the machine to life.

Longer programs include:

- Music While You List
- Happy Christmas Tree
- Permanent Screen Clock
- Spin the Screen
- Simple Machine Code Game
- Ladder Graphics

There is no need for any technical knowledge or maths. This book is for the enthusiastic beginner to assembly programming.

Most of the book is also suitable for assembly programming on the Electron.