# ASSEMBLY LANGUAGE PROGRAMMING for the BBC MICROCOMPUTER

Ian Birnbaum

# Assembly Language Programming
# for the
# BBC Microcomputer

**Macmillan Computing Books**

*Assembly Language Programming for the BBC Microcomputer* Ian Birnbaum

*Advanced Programming for the 16K ZX81* Mike Costello

*Microprocessors and Microcomputers — their use and programming* Eric Huggins

*The Alien, Numbereater, and Other Programs for Personal Computers – with notes on how they were written* John Race

*Beginning BASIC* Peter Gosling

*Continuing BASIC* Peter Gosling

*Program Your Microcomputer in BASIC* Peter Gosling

*Practical BASIC Programming* Peter Gosling

*The Sinclair ZX81 – Programming for Real Applications* Randle Hurley

*More Real Applications for the Spectrum and ZX81* Randle Hurley

*Assembly Language Assembled – for the Sinclair ZX81* Tony Woods

*Digital Techniques* Noel Morris

*Microprocessor and Microcomputer Technology* Noel Morris

*Understanding Microprocessors* B. S. Walker

*Codes for Computers and Microprocessors* P. Gosling and Q. Laarhoven

*Z80 Assembly Language Programming for Students* Roger Hutty

# Assembly Language Programming
# for the
# BBC Microcomputer

Ian Birnbaum

*First published 1982 by*
THE MACMILLAN PRESS Ltd
London and Basingstoke
Companies and representatives
throughout the world

*Printed in Great Britain by Unwin Brothers Limited,*
*The Gresham Press, Old Woking, Surrey.*

ISBN 0 333 34585 1

Dedicated to Theresa

# Contents

# *Preface*

Every BBC Microcomputer, whether Model A or Model B, comes equipped with an immensely powerful and very fast assembler. What is more, assembly language statements and BASIC statements can be freely mixed, hugely increasing the programmer's potential control over the machine.

This book shows you how to establish that control. It assumes that you are proficient in BASIC, for if you are not this is probably not the best time to learn assembly language. But it assumes no knowledge of assembler at all, taking you step by step from the basics to their complex implementation.

Since every user of the BBC Microcomputer assembler will have a working knowledge of BASIC, it is possible to use that knowledge to motivate and illustrate the ideas in assembly language. This book takes that approach, and this should help you to master assembly code, for you will always be acquainted with the fundamental concepts by seeing their connection with BASIC.

I had three types of readers in mind in writing this book. Firstly all current owners of BBC Microcomputers who want to extend their knowledge into machine code. To help them with self-instruction, this book contains a considerable number of exercises and *a full solution is provided for every one*. Secondly, the teacher or student of Computer Science who wants to use this text in a structured course. The book is the result of many years' teaching experience, and it is designed according to a teaching strategy which the author has found to be very successful.

And thirdly, those people, already experienced BASIC programmers, who are wondering whether to buy the BBC Microcomputer, when there seems so much competition from cheaper and seemingly comparable computers. This book should help to convince them that the BBC Micro is worth the extra expense. Quite apart from its superb BASIC, the Micro possesses an assembler which turns it into a potential 6502 development system in its own right! It also possesses an operating system which is designed to mesh with assembly language programming in an extraordinarily simple way. One of the aims of this book is to show you how to exploit these features to the full.

The book contains 73 listings of programs, many of which will be found to be useful utilities in their own right, quite apart from their value in teaching you assembly language. In particular it contains a full machine code monitor, a suite of machine code sorting programs which you can use on BASIC variables, a high resolution screen copy to the Epson printer and a program compactor. There are two companion tapes available with the book if you do not feel you want to type in the programs yourself. Each tape also contains two extra programs: the first, a universal graph plotter and a 6502 disassembler which displays in standard 6502 mnemonics. The second contains two utilities for programming: a machine code program which will find the locations of any segment of code in a BASIC program (equivalent to the FIND command found in some utility packages); and another machine code program which will replace any segment of code in a BASIC program by any other segment; so, for example, you can change any variable's name in the whole program in an instant.

The book is completely self-contained: full information on the 6502 instruction set is provided throughout and summarised in an Appendix. Other Appendices cover floating point and the user port, and there is a section on combining programs in the BBC Computer using PAGE and *LOAD.

Thanks are due to Mrs Barry who typed up a very untidy manuscript quickly and efficiently, and with very few errors.

Ian Birnbaum
Needingworth
May 1982

# Chapter 1 Preliminary ideas

## 1.1 WHAT IS A COMPUTER?

In its simplest form, a computer can be considered in four sections: *input, output, microprocessor, internal memory.* Figure 1.1 shows the interrelationship:

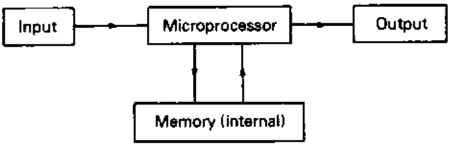


*Figure 1.1: Simplified diagram of a computer*

Most *input* is through the keyboard, but other input devices and channels include: casette tape system, disc system and sensor devices connected to an input port.

Most *output* is through the TV or monitor (the VDU), but other output devices and channels include: casette tape system, disc system, printer and control devices connected to an output port.

Notice that the cassette and disc systems are both input and output devices. These are sometimes referred to as backing store. Printers and suchlike are sometimes referred to as peripherals, things outside the main system.

All input must pass through the *microprocessor,* at least at some time or another. It may reside temporarily in some internal memory (often referred to in this context as a buffer or latch), but the microprocessor will deal with it when it can. Similarly all output will be directed by the microprocessor and all will pass through it. The microprocessor is the 'brain' of the system, and this book is concerned with how to program it directly. There are also other 'lesser brains' to be found in a computer, but these are not usually under our control and correspond roughly, following the metaphor, to the autonomic nervous system.

Internal memory can be divided into two parts: *ROM* (read only memory) and *RAM* (misleadingly called random access memory - read/write memory is a better name). ROM contains information which is fixed: the microprocessor is unable to modify it. Usually it contains instructions and data which the microprocessor will need in the same form when a specific task is demanded. In particular, in the BBC micro it contains the Operating System (OS) and the BASIC Interpreter. A considerable advantage of ROM is that information in it does not disappear when the computer is switched off.

Information in RAM will, by contrast, disappear if power is removed. It has the advantage, however, of allowing information to be modified. It is in RAM that all the programs and data we input will reside.

Some RAM is given a special function. Some, for example, will be used by the microprocessor as a sort of 'scratch pad' (this is called the stack and is covered in Chapter 9). Other parts are reserved for the OS or the BASIC Interpreter in which to store results and information. And some parts are connected to input/output channels. RAM used in this way is often referred to as *memory mapped*. For example, the BBC micro has a memory-mapped VDU, each character on the screen corresponding to a specific portion of memory which is fixed for each graphics mode chosen (this is not strictly true, as we shall see in Chapter 9).

There are some specialised chips in the computer that act as RAM, though they are not usually referred to in this way. The best example of these are the input/output chips, going under a variety of names (*PIO* - programmable input/output, *PIA* - peripheral interface adapter, *VIA* - versatile interface adapter). PIO is the most descriptive: the chip consists of a series of memory locations in which data passing in or out can be latched; certain locations contain information on whether a particular channel is to be conceived as input or output, and this information can be changed by the microprocessor. Although these chips contain more than just memory (for example some contain a timer, results of which are available at a specific location), they are most conveniently thought of as RAM because they are *addressable,* an idea to which we now turn.

## 1.2 HOW IS MEMORY ORGANISED IN A COMPUTER?

If we want to refer to a specific location of memory, it will need to have a name. Since we may want to refer to any part of memory at some time, we must make sure that memory is organised in such a way that every location has a name and that this name is unique.

When computers are built they are wired up in such a way that the microprocessor can refer to a specific location by outputting a series of pulses called *an address*. The set of wires through which they pass is called the *address bus*.

Since each wire will either have a pulse or not have a pulse there are two states which we may label 1 (pulse) and 0 (no pulse). A microprocessor is a *digital device* because it always understands things and communicates with other parts of the computer in this *two-state* way. Thus: a switch is either on or off; an element either has a positive or negative field etc.

The microprocessor in the BBC micro can accept 16 wires on its address bus. Since each wire can either be 1 or 0 this gives a total of $2^{16}$ = 65536 addresses, that is from 0000000000000000 to 1111111111111111. Hence there can be at most 65536 locations of memory.

Now writing 16 ones and/or zeros like this is very hard to read, and so we will adopt a notation that makes it easier. We will divide our 16 digits into four groups of four. So for example in 0111101101000010 the four groups are:

0111 1011 0100 0010

Now each group of four can have one of sixteen different forms, from 0000 to 1111 ($2^4$). We can use the numerals 0 to 9 for the first ten; after that we will use A, B, C, D, E, F. Table 1.1 gives the details. In that table we can conceive of the four digits on the left as the display on a rather odd car odometer (the meter measuring the distance covered). Each cog on the dial has just two numerals, 0 and 1. As the cog revolves through one revolution it pushes the next one on half a turn. In this way, the first sixteen numbers (0-15, decimal) are generated in the order shown.

*Table 1.1: Relationship between binary and hexadecimal*

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

Using this notation we can write the number above as 7B42. The number 0010010101000000 will be written 2540, but this is rather misleading because it looks like two thousand five hundred and forty, which it isn't. So to prevent confusion, we precede these hexadecimal numbers, as they are called, by the ampersand sign (&). Thus we write them as &7B42 and &2540 (some computers use the dollar sign, $, but not the BBC micro).

You will notice in Table 1.1 the terms *binary* and *hexadecimal*. Binary means two (there are two possible numerals, 0 and 1) and hexadecimal means sixteen (there are sixteen possible numerals 0 to F). In the same way decimal, our usual system of representing numbers, means ten (numerals 0 to 9).

Many books spend ages explaining how to convert from one system to another, but this is a complete waste of time. Your BBC computer will do it for you.

For example, type into your computer

```
P. &7B42
```

and see what you get. This is the decimal equivalent of &7B42.

Similarly, type in

```
P. ~14321
```

and see what you get. This is the hexadecimal equivalent of 14321.

If you wish you can write a program to convert either way but it is hardly worth it; you might as well operate in direct mode, as above. It is worth experimenting a little with various numbers to see the equivalence operating for yourself.

— || —

It should be clear to you by now why it is convenient for the computer to work in binary and why it is convenient for us to work in hex (the usual abbreviation for hexadecimal). From now on we will think of all the memory locations in terms of hex.

Now in order to make the wiring as simple as possible, a simplifying concept called *pageing* is used. The 65536 addresses of memory are conceived as a series of *pages*. The page number is given by the top two digits of the hexadecimal number; the bottom two digits give the location in that page. The best image is that of a book with 256 pages, each page having 256 lines, each line being a memory location. Rather eccentrically, the book's first page is labelled zero, and is called *zero page*. It is a very important area of memory as we shall see in the next chapter. Figure 1.2 should make the idea clear. Thus address &F1B2 refers to location 178 in page 241; that is, address number 61866. We can think of this in the following way: the high byte &F1 accesses all the locations in page 241; then the low byte &B2 picks out a particular location in that page, location 178. The top half of the address bus is thus wired up to access pages, and the lower half to access one of 256 locations in any page.

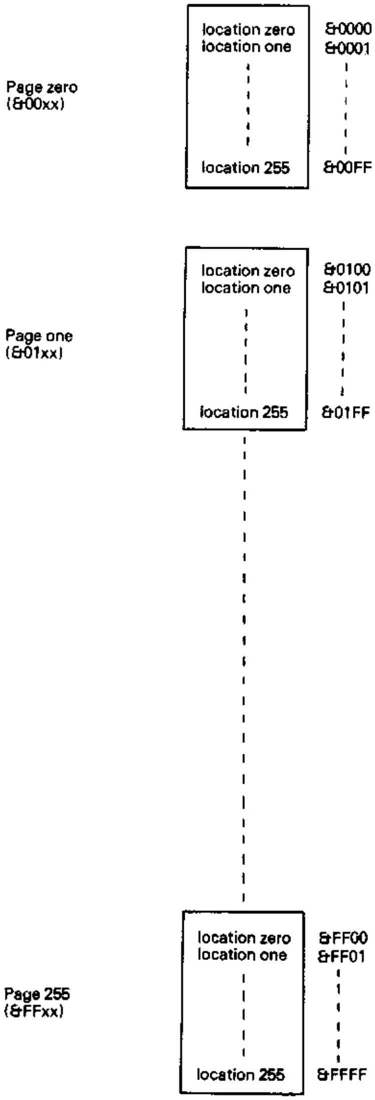


*Figure 1.2: Pageing*

When referring to memory en masse it is conventional to work in units of 4 pages and refer to this as a *K of memory* locations. Thus the microprocessor in the BBC micro can address 64K of memory locations; your machine will either have 16K of RAM (model A) or 32K of RAM (model B).

So far we have talked rather vaguely about a memory location; we must now ask what we can put in such a location.

As we have already said, the microprocessor only outputs ones or zeroes and can only understand ones and zeroes. Hence it comes as no surprise to find that what exists in these locations is a series or ones and zeroes.

The microprocessor in the BBC machine communicates with the *contents* of a memory location through a set of eight wires called a *data bus*. Since there are eight wires, this allows any one of $2^8$ or 256 different numbers in one location i.e. from &00 to &FF.

We have referred so far to an 8-wire data bus and a 16-wire address bus, but the term usually used is *bit* not wire (bit standing for *b*inary dig*it*). Thus the data bus is 8-bit and the address bus 16-bit. A bit is simply either a 1 or 0. With their usual good humour, computer scientists have called a memory location consisting of eight bits a *byte* (you will meet the nybble later, so be warned!)

Thus a byte consists of 8 bits in sequence. These bits are usually numbered from right to left, zero to seven, the seventh being called the most significant bit (MSB) and the zeroeth the least significant bit (LSB); see Figure 1.3.

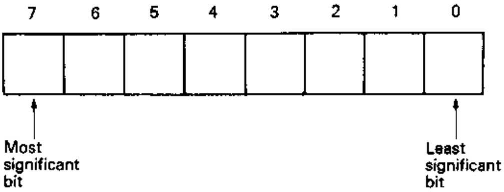


*Figure 1.3: Diagram of a memory location, a byte.*

We can say, then, that the microprocessor can address 64K bytes, but that the memory contains 512K bits. The memory is therefore *byte-addressable* (i.e. each byte can be addressed) but not *bit-addressable* (i.e. an individual bit cannot be addressed except by addressing the byte of which it is a member).

## 1.3 HOW IS THE 6502 MICROPROCESSOR ORGANISED?

The microprocessor in the BBC micro is a 6502 manufactured by MOS Technology. It is an 8-bit microprocessor in that it can only accept and transfer 8 bits of data at a time.

The diagram in figure 1.4 is a very simplified block diagram of the organisation or *architecture*, as it is usually called, of the 6502 microprocessor.

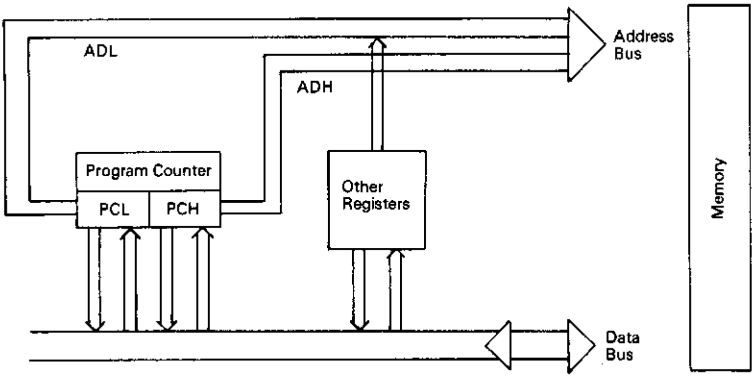


*Figure 1.4: Partial block diagram of 6502 architecture*

In future chapters the box labelled 'Other Registers' will be opened out to reveal further aspects of the architecture when they are needed. By Chapter 9 all the architecture will have been revealed. For now, we should note the difference between a *register* and a RAM location. The essential point is that a register is internal to the microprocessor and can be accessed without having to output an address on the address bus. As a consequence accessing registers is relatively faster than accessing other memory locations.

We shall focus on the *program counter* at this stage. In order to operate, the microprocessor needs a sequence of instructions to follow. These instructions will reside in memory either in ROM or in RAM. If they are in ROM they will have been deposited there by the Computer's designer (in the BBC micro's case, Acorn); if they are in RAM they might well have been put there by you.

The computer needs to know where the first instruction is to be found. The address of the memory location where this instruction lies is put in the program counter: in PCL goes the low byte of the address and in PCH the high byte. These bytes are sent out on the address bus and back comes the instruction on the data bus.

But what does an instruction look like? We have already said that all that can be sent on the data bus are a set of 8 bits. As long as the microprocessor expects to get an instruction it will decode the particular 8 bits in a special register (the instruction register and decoder) and respond to the instruction according to the instruction set details fixed in the circuitry of the microprocessor. It is these instructions we will be looking at in the rest of the book.

After it has decoded the first instruction it will know what to expect next. The program counter will automatically increment by one and put out the next address in sequence on the address bus.

Back will come the next byte on the data bus. How this byte is interpreted will depend on the instruction previously decoded, as we shall see in later chapters.

The function of the program counter, then, is to fetch, one by one, a sequence of bytes from a section of memory, the starting address of which is put in the program counter initially. You may be wondering how this initial address gets there, since the microprocessor needs an instruction to put it there in the first place! The answer is that there is a special line connected to a pin of the microprocessor called a *reset line*. When a pulse is sent to this line, as it is when the computer is turned on, the microprocessor is pre-programmed to obtain an address from a fixed location (&FFFC for PCL and &FFFD for PCH) in which the first instruction is to be found. These addresses are in ROM in the BBC micro (as they must be) and the designers of the computer have written a special start-up program, the first instruction of which is at the address contained in &FFFC and &FFFD. A reset is also generated if the 'break' key is pressed.

Notice finally that the program counter is connected not only to the address bus but to the data bus. This is because the counter needs to be set initially and on occasion may need to be saved in memory. We shall have more to say about this in Chapter 9.

## 1.4 MACHINE CODE AND ASSEMBLY LANGUAGE

We have seen that a microprocessor can only understand a set of ones and zeroes; that is, information at its level is expressed in bits. We have also seen that an instruction is received through the data bus as a set of 8 bits, a byte. A sequence of bytes, stored consecutively in memory, can be accessed by the microprocessor using the program counter and constitutes *a program*. This program is said to be in *machine code,* since it is in the form directly comprehensible to the microprocessor.

It is not directly comprehensible to humans however. &8D as an instruction does not mean much to us; we would have to continually look up tables or else commit the entire set of instructions to memory. And this would lead to errors.

In the early days of computing a brilliant but now commonplace insight occurred: why not write a program which would allow humans to communicate with computers in a more accessible way. This gave rise to the idea of *assembly language*. Each instruction is given a *mnemonic,* an 'aide memoire', and the programmer can work solely with these mnemonics. So, for example, &8D becomes STA, which is easy to remember. It means 'store a copy of the accumulator somewhere', an idea we will meet in the next chapter.

Thus a program already written and stored in ROM in the computer, called an *assembler,* translates each mnemonic into its machine code equivalent. Some assemblers are more powerful still and allow the programmer to use variables as in Basic; the BBC micro's assembler is like this. In the following chapters we will be learning to program in this assembly language.

There are a few concepts connected with assembly language which it is worth mentioning now. We have already met the idea of a mnemonic. The instruction this represents often needs to act upon some information contained in the next byte (or sometimes the next two bytes). In such cases, the next byte (or bytes) is called the *operand field* or usually just the *operand*. (This is an ugly Latinate word, and not one I particularly like, but it is in common usage and so needs to be known.) In effect it means that a byte or bytes are being understood by the microprocessor as an item of data or as an address of where some data is stored, rather than as an instruction. We will expand upon this idea in the next chapter.

We have already mentioned that a program written in assembly language needs to be translated by the assembler. It is usual to call the program to be translated (i.e. the one written in assembly language) the *source code* and the machine code program corresponding to it (i.e. the one produced when the assembler makes the translation) the *object code*. The object code is thus just the machine code program corresponding to our assembly language program. Figure 1.5 should make this clear.

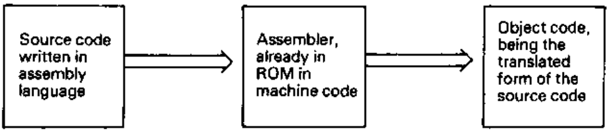

Source code written in assembly language
Assembler, already in ROM in machine code
Object code, being the translated form of the source code

*Figure 1.5 Translation of source code into object code by the assembler*

## 1.5 COMPILERS AND INTERPRETERS: WHY USE ASSEMBLY LANGUAGE?

Assembly language is a vast improvement on machine code but it is still a very *low-level language*. By this we mean that each mnemonic in assembly language corresponds to one and only one machine code instruction. By contrast, if we write in a *high-level language*, each high-level instruction will correspond to a whole set of instructions in machine code.

High-level language translators fall into two groups: *compilers* and *interpreters*. A compiler takes the source code written in the high-level language and translates it in object code. This object code may be itself machine code or may be what is called a pseudo-code which will itself be translated into machine code. The important point here is that once the object code is produced the source code is no longer necessary for the running of the program.

By contrast, an interpreter cannot dispense with the source code. A program written in BBC BASIC for example, needs to reside in the computer memory while the program is being run. If BBC BASIC were a compiled language, once the translation was over, we would remove the BASIC program and use the object code instead. This is because in a compiled language translation and execution take place at quite separate times. In an interpreted language, however, each statement is first translated and then executed. Thus every time the interpreter meets the statement GOTO 100 it needs first to translate this into machine code before executing it. The advantage which an interpreted language has over a compiled language is that it is much easier to edit and debug programs. The disadvantage is that programs run more slowly.

The advantage of high-level languages in general over assembly languages is that it is easier to learn a high-level language and writing a program is both simpler and quicker to do than in assembly language. Also, assembly is specific to the microprocessor used: the 6502 assembly code you will learn in this book will not work on a Z80 microprocessor for example. High-level languages are generally portable between machines, although different machines will need different compilers or interpreters.

Why then use assembly language at all? There are four principal reasons: programs written in assembly language will generally be more efficient and shorter than those written in a compiled language; they will be very much faster to execute than those written in an interpreted language like BBC BASIC; working with assembly language gives you great insights into the operations of a computer and allows you to exploit some of its special features not easily accessible at high-level; and, last but not least, it is fun!

However, one must not underestimate the difficulties of writing even medium-sized programs in assembly language. For this reason, while it is fun to write in assembly language, it is not worth attempting a medium-sized project entirely in assembly language unless one or both of the two middle reasons above are of particular importance in your chosen application. This is because the BBC computer allows you to mix BASIC and assembly language in a particularly easy way. This means that you can use assembly language to do things which need to be done quickly or which cannot be realistically accomplished by BASIC or which will need to be accomplished when the BASIC interpreter is not in use, and that you can use BASIC to do all those things for which assembly language is unnecessary.

There is nothing to be gained in using assembly language when BASIC will do (except perhaps a rather peculiar one-upmanship). Nevertheless, for pedagogical reasons, I have used some examples in this book which could be programmed in BASIC without too much loss; and again for pedagogical reasons I will use BASIC statements to motivate the introduction of new instructions in the 6502 instruction set. But, where possible I mix BASIC and assembly language in programs in proportion to their relevance to the particular task. Indeed the great advantage of the BBC micro lies in this freedom to mix the two languages; and one of the aims of this book is to show you how to exploit that freedom.

## Exercise 1

1. The following concepts have been introduced in this chapter. Try to write a brief description of each one:

ROM; RAM; memory-mapping; PIO; addressable memory; address bus; two-state device; binary numbers; hexadecimal numbers; pageing; zero page; data bus; bit; byte; 64K bytes; bit- and byte-addressability; register; program counter; reset; machine code; assembly language; mnemonic; operand; assembler; source code; object code; low-level language; high-level language; compiler; interpreter.

2. Why is it that a memory-mapped VDU cannot be in ROM and that the address of the start-up instruction cannot be in RAM?

3. If it is generally regarded as a disadvantage to have an 8-bit data bus instead of a 16-bit data bus, can you think of reasons why the 6502 microprocessor was not designed with a 16-bit data bus?

4. Referring to figure 1.4 explain why the data bus is bidirectional (i.e. data can pass in both directions) but that the address bus is unidirectional.

5. Why is a program executed under an interpreter slower than one compiled and then executed?

# Chapter 2 Assignments

## 2.1 THE ACCUMULATOR

The first and the most important of the 'Other Registers' in figure 1.4 we will consider is the *Accumulator.* This is an 8-bit register, like all the 'Other Registers', and communicates with both the address bus and the data bus. The principal feature of the accumulator is that all arithmetic and most logical operations must take place using it. So, for example, if two numbers are to be added, one of them must be in the accumulator. The result of this addition will also be placed in the accumulator.

We will cover these arithmetical and logical operations in the next chapter, but in this one we will concentrate on the other function of the accumulator: as a temporary storage location used when moving data from one memory location to another. The 6502 does have other registers for this purpose as we shall see in Chapter 5, but the accumulator is the most important of these. We can view the accumulator, then, as an interim storage location inside the microprocessor, being the place through which movements of data pass and where intermediate results are stored.

Let us look now at some of the simplest assignment statements involving the accumulator. To motivate ideas in this and some of the later chapters we will try to find assembly language equivalents to some common BASIC statements.

## 2.2 WHAT IS THE ASSEMBLY LANGUAGE EQUIVALENT OF LET NUM1 = 17?

In BASIC, this statement directs the computer to put the number 17 into a storage location labelled NUM1. What is the equivalent in assembly language? Consider the following statements in assembly code:

```
LDA #17

STA NUM1
```

This does not quite suffice, as we shall see in a moment, but let us first examine what this code does.

LDA is an instruction which means 'load the accumulator' and #17 means 'with the number seventeen'.

STA is an instruction which means 'store a copy of the accumulator' and NUM1 is the name of the storage location where it is to be stored. So taken together, these statements store the number 17 in a location labelled NUM1, via the accumulator.

The problem is, however, that we need to tell the computer *where* this storage location called NUM1 is. In BASIC this is not necessary: the interpreter, a complex machine code program already in the computer, sorts this out for us. We don't need to tell the computer where NUM1 should go; and it doesn't need to tell us where it has put it. (In fact it puts it into five bytes, not one, but this need not concern us now.)

In assembler things are different, however. Without a directive from us, the computer will not know where NUM1 is. As it stands, then, the computer will give us an error message telling us that the label NUM1 is undefined.

We define it by preceding the first line with:

```
NUM1 = &0DFF
```

The computer will now store 17 in the location &0DFF. We must be careful in our choice of memory location that it cannot be contaminated by other operations in the computer. If necessary we may have to protect the area of memory by suitably redefining portions of memory. We will have more to say about this later in this chapter.

Even with this protection, it may well be that our choice for NUM1 is not the best. The 6502 is designed to give special preference to memory locations in page zero (which, as we saw in the last chapter, is between &0000 and &00FF). To understand how this gains, we need first to understand how our simple program is stored in the computer.

We have to tell the assembler where the first instruction will go. We do this by writing

```
P% = &0D00
```

at the beginning (again making sure this is a protected area). When we now get the assembler to translate our mnemonics into machine code, the very first instruction goes into memory location &0D00. Let us see what the machine code translation looks like.

LDA #17 takes up two bytes: &A9 in &0D00 and &11 in &0D01. &A9 is the *Op Code,* as it is called, for LDA when the number to be loaded is contained in the next byte. This is signified by the symbol # in the assembly code, and is called, for obvious reasons, *immediate addressing*. The number to be loaded is immediately available; it doesn't need to be fetched from another part of memory first.

When NUM1 = &0DFF, STA NUM1 takes up three bytes: &8D in &0D02, &FF in &0D03 and &0D in &0D04. Notice that the lower byte of the address is stored first; fortunately the assembler does this automatically so we need not worry too much about it now. This sort of addressing, where a two byte address indicating where the contents are to be found follows the instruction, is called *absolute addressing*.

By contrast, when NUM1 = &70, say, STA NUM1 takes up only two bytes: &85 in &0D02 and &70 in &0D03. Only the lower byte, &70, of the address needs to be supplied; the computer knows that the higher byte is &00. Note that the Op Code, &85, is different from &8D, the one above. This reflects the different mode of addressing. Here, the addressing mode is called *zero page addressing*. Fortunately again, the assembler sorts all this out, so we do not need to change the mnemonic code in any way for zero page addressing. STA NUM1 is coded &8D, &FF, &0D if NUM1 = &0DFF, and &85, &70 if NUM1 = &70, quite automatically by the assembler.

Hence by using a zero page location we save one byte of memory. This can be crucial in large programs. Even more importantly, the zero page operation is faster than the absolute one; and again this can be crucial.

So to summarise: if our symbolic code is this:

```
  P% = &0D00
NUM1 = &0DFF
 LDA   #17
 STA   NUM1
```

it goes into memory thus:

| Location | &0D00 | &0D01 | &0D02 | &0D03 | &0D04 |
|---|---|---|---|---|---|
| Contents | A9 | 11 | 8D | FF | 0D |

If we change line two to NUM1 = &70, it goes into memory as:

| Location | &0D00 | &0D01 | &0D02 | &0D03 |
|---|---|---|---|---|
| Contents | A9 | 11 | 85 | 70 |

## 2.3 MORE ON THE IMMEDIATE, ABSOLUTE AND ZERO PAGE ADDRESSING MODES

In the last section we met three modes of addressing. These ideas are so important that it is worth elaborating upon them further.

We have seen in Chapter 1 that when the microprocessor first receives a byte which it expects to be an instruction, it decodes it and then acts upon it. Whether it interprets the next byte as a piece of data, a zero page address or the least significant byte of a non-zero-page address depends on the particular instruction it has received. Thus, the address mode information is contained in the Op-Code of the instruction. This is a crucial aspect of the 6502's operation and it must be thoroughly grasped.

The assembler obscures this fact somewhat, since the same mnemonic is used for all the address modes. Thus LDA #17, LDA &0070, LDA &0DFF all use the mnemonic LDA. The # in the first case means immediate addressing (op code &A9) where the next byte is to be treated as data. In the second case (LDA &0070), because the address is in zero page it will be coded as zero page addressing (op code &A5), and the next byte is treated as a zero page address. In the last case (LDA &0DFF), the address is not in zero page so it will be coded as an absolute address (op code &AD) and the next byte will be the lower byte of this address. In this case the following byte will be interpreted as the higher byte of the address; by contrast, in the first two cases this following byte will be interpreted as a new instruction.

The use of a single mnemonic for different address modes is a relief to our human memories but we must be aware that depending on the operand it is encoded in different ways in the computer's memory.

## 2.4 WHAT IS THE ASSEMBLER EQUIVALENT OF LET NUM2 = NUM1?

We assume that a value has already been put into NUM1, a zero page location. We code this further statement in assembler as follows:

```
LDA NUM1
STA NUM2
```

Of course we need to tell the assembler the address of NUM2. If we can, we should again choose zero page, say &71. Putting everything together we have:

```
  P% = &0D00
NUM1 = &70
NUM2 = &71
 LDA   NUM1
 STA   NUM2
```

When translated into machine code by the assembler, the following is produced:

```
Location   &0D00   &0D01   &0D02   &0D03
Contents    A5      70      85      71
```

Notice that LDA is coded as &A5 and not &A9 as before. This reflects the change in addressing mode, here zero page, before, immediate. In this case the number to be loaded, 17, has to be fetched from the memory location &0070. The computer puts the address &0070 out onto the address bus, and back comes the contents, 17, on the data bus. All this is initiated automatically by the Op Code &A5.

There is an ambiguity in the BASIC statement which needs to be resolved. When we write LET NUM2 = NUM1 we usually regard NUM2 and NUM1 as the name of variables, just as we do in algebra. Hence it is often conceptualised as equivalent to let y = x, but this is not in fact what it means. The precise meaning of this statement is: assign to memory location NUM2 the contents of memory location NUM1. This may seem to be splitting hairs, and in many practical applications of BASIC perhaps it is, but when using assembly language the distinction is crucial, although the ambiguity remains. So, LDA NUM1 means load the accumulator with the *contents* of memory location NUM1, whereas STA NUM1 would mean store a copy of the contents of the accumulator into memory location NUM1. The same label is used in two different ways: in the first case NUM1 refers to the *contents* of a memory location, in the second NUM1 refers to the *address* of a memory location. There is no *real* ambiguity here, however, since the mnemonics LDA and STA provide the distinction is meaning.

A useful way to resolve real ambiguities though is to put brackets round a label if we mean to refer to its contents: thus (NUM1) refers to the contents of the memory with address NUM1. When we wish to make this distinction clear in our *discussion* of the various instructions we are quite free to use the convention, and this we will do. Moreover, the 6502 assembler does use the convention itself on certain occasions, as we shall see later, but it makes very specific use of it and so we are *not* free to put brackets round our labels when *writing* assembly language.

Note finally that we do not have to use labels like NUM1 and NUM2 at all. We could instead write directly:

```
LDA &70
STA &71
```

At this stage it does not matter, but as our programs become more complex, the use of labels becomes more convenient and important.

## Exercise 2

1. State which form of addressing is used in each case. Assuming the first instruction in each statement begins at &0D00, show how

the assembler would translate each statement into machine code.

(a) LDA #14

(b) LDA &7F40

(c) LDA &20

(d) STA &7A72

(e) STA &00

(f) LDA #&12

(g) STA &0002

(h) LDA 14

(i) STA 1024

2. If we combine the two programs in Sections 2.2 and 2.4 into a single program, explain why LDA NUM1 can be omitted.

Write out the program in assembly code, and show what the machine code translation would be.

3. Code this BASIC program into assembly language (assume NUM2 and NUM3 already have values).

```
LET NUM1 = NUM3
LET NUM3 = NUM2
LET NUM2 = NUM1
```

## 2.5 WHERE TO PUT MACHINE CODE PROGRAM IN THE BBC COMPUTER

We have already seen that we have to tell the assembler where our first instruction will go: we do this by setting the variable P% equal to some value. There are principally six ways of doing this:

### (a) Putting the machine code above the BASIC program

As we shall see in a moment, assembly language programs are entered into the machine in BASIC programs. We could put the machine code translation of the program above this program. There are four ways to do this:

(i) P% = TOP + 1000. TOP is a BASIC function which gives the location following the last memory location occupied by the program text. By adding 1000 to it we reserve adequate space for any dynamic variables used in the program.

(ii) LOMEM = LOMEM + 250. LOMEM is a BASIC 'pseudo-variable' which controls where the first dynamic variable is to be placed. By setting it to 250 above its default value (which is TOP), we reserve 250 bytes for our machine code program. This statement must precede all references to dynamic variables (i.e. any except α% and A% - Z%). We can then write P% = TOP.

(iii) DIM P% 250. This reserves 250 bytes in the dynamic variable region above the program text and automatically sets P% equal to the address of the first location of this reserved block

(iv) HIMEM = HIMEM-250. HIMEM is a BASIC 'pseudo-variable' which specifies the highest point in memory available to any program. Usually it is set to the first location of screen memory (e.g. &3C00 in Mode 7 on a Model A). BASIC programs use the top of memory to store information about procedures and functions etc., so we cannot just put a machine code program at the top of memory. However, by resetting HIMEM, we reserve 250 bytes at the top of memory for our program. This statement should be the first one in the program. Note that changing from a smaller to a higher resolution mode (e.g. MODE 7 to MODE 4) will destroy the machine code program.

### (b) Putting the machine code below the BASIC program

(i) PAGE = PAGE + 256. PAGE is a 'pseudo-variable' which controls the starting address of a BASIC program. It is always a multiple of 256 (reference to Section 1.2 will remind you why), and its default value is &E00 on the ordinary model A or B (i.e. a system without discs etc.). This instruction should be performed *before* loading the program. We can then safely set P% to PAGE-256 in the program.

(ii) Using memory below &E00: The locations from &900 to &AFF are used by the cassette system when the OPENIN and OPENOUT commands are used but not by the system to save and load programs. They are also used if the RS423 port is in operation on a model B. Hence, in certain circumstances these locations may be safe.

The locations &C00 to &CFF are reserved for programmed characters (ASCll codes 223-255 inclusive). If you do not expect to use these at the same time as a machine code program then this area is free.

Finally locations &D00 to &DFF seem to be untouched by the operating system, at least in the standard models A and B. This means that this area is safe for machine code programs on the standard models.

Other areas are best left alone. &B00 to &BFF, for example, is used for the programmable keys, and even if you do not intend to use them, accidentally pressing one with machine code in this area can have strange effects. Even stranger things happen if you put code into the sound buffer at &800 to &8FF!

---

Of these six options for machine code location, (a)(iii) is useful when a machine code program is to be used in conjunction with a BASIC program, since they are kept together in a simple way. Wherever the BASIC program is loaded into memory, the machine code will run. Moreover, unlike most other micros, the BBC micro allows us to store many programs in memory at the same time, using the PAGE command. Hence, we can use a machine code program contained in one BASIC program in a second BASIC program (see Appendix 6 on the use of PAGE).

(b)(ii) is most useful for stand-alone machine code programs. Clearly &D00-&DFF is the safest area, but in many cases if more than 256 bytes are needed &C00-&DFF is reasonable. The only disadvantage here might be the effects when non-standard model B's are used (with discs etc.).

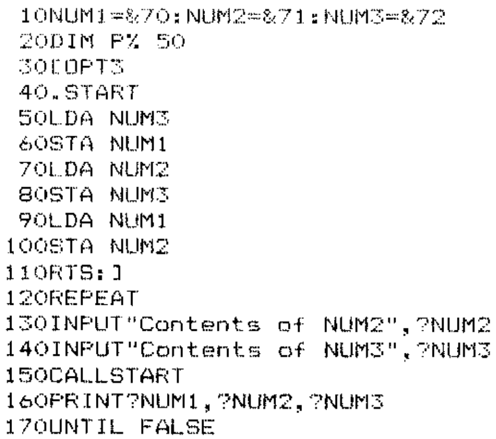
In this book option (a)(iii) will be followed in the earlier chapters, where BASIC and machine code are often linked, usually for pedogogical purposes. In later chapters, and especially Chapter 10, option (b)(ii) with &D00 will be used. The one exception to this is the Monitor program which requires &330 bytes: in this case option (a)(iv) will be used.

Finally in this section, note that the designers of the BBC computer have reserved 32 zero page locations for assembly program use. This is very useful, for we have already seen the saving involved in using zero page. These locations are from &70 to &8F inclusive.

## 2.6 HOW TO INPUT ASSEMBLY LANGUAGE PROGRAMS INTO THE BBC COMPUTER

Listing 2.1 shows how to input the solution to Q.2, Exercise 2, into the computer. Put it in and run it. Try different numbers and convince yourself it works; press ESCAPE when you have tried a few.

LISTING 2.1



```
   10NUM1=&70:NUM2=&71:NUM3=&72
   20DIM P% 50
   30[OPT3
   40.START
   50LDA NUM3
   60STA NUM1
   70LDA NUM2
   80STA NUM3
   90LDA NUM1
  100STA NUM2
  110RTS:]
  120REPEAT
  130INPUT"Contents of NUM2",?NUM2
  140INPUT"Contents of NUM3",?NUM3
  150CALLSTART
  160PRINT?NUM1,?NUM2,?NUM3
  170UNTIL FALSE
```

Let us look at the listing in detail. In line 10, we give zero page addresses to the three labels we shall use. In line 20 we set the program counter as recommended in the last section. As a rule of thumb, allow 3 bytes for every instruction. This will always be ample, and will usually give some room for any editing and insertion required later. There is no need to count the instructions, though; a rough estimate is all that is required, and rounding up to the nearest fifty will do. Line 30 is the beginning of the assembly code: the open square bracket, [, tells the interpreter that. OPT3 is what is called a pseudo operation: it is not strictly part of the 6502 assembly language. Its function is to specify whether and how the translation into machine code will be listed on assembly. In this case, 3 indicates that the translation will be listed, and any errors displayed (the possible errors at this stage are syntax error or unknown label).

In line 40 we begin with .START. This label will automatically be given the address into which the first instruction will be put. Any label will do, as long as it begins with a full stop, and otherwise conforms to all the rules for BASIC variables. (Some assemblers restrict the length to six, but not the BBC micro assembler.) Clearly START is a sensible label, and we should endeavour to use labels as meaningful as possible. We shall have much more to say about labels like these having the function of referring to an address within the program, in Chapter 4.

The assembly program proper is in lines 50 to 110. The assembly program is translated into machine code line by line through to line 110. In line 150 control is passed to the machine code program, located at the address referenced by START. The function of the new instruction RTS in line 110 is to return control to the statement following this call, when the machine code program has finished. Every assembly program must end with RTS. We shall have much more to say about RTS in Chapter 9, but for now we regard it as an essential terminator to the assembly program. The close square bracket, ], indicates to the interpreter that the assembly program has finished (note the colon before it, essential if it does not begin a line).

In lines 130 and 140 we put values into address locations NUM2 and NUM3 prior to calling the machine code program, and in line 160 we print out the values in the three address locations after returning from the machine code program. The query operator ? means the 'contents of the memory location whose address is the value of the variable attached'. Thus P.?NUM2 prints the contents of the memory location with address NUM2 i.e. the contents of &71.

It is possible to extend the notation to a binary function and write NUM?1 instead of ?(NUM+1), but this is less readable, in my view- and saves very little.

Using the query operator, we can pass values to the machine code program and output results from that program. It is possible to do this directly from the machine code program - essential if speed is an important factor - but the details of this must wait until Chapter 9.

It is important to understand that when lines 10 to 110 of the program are run all that happens is that the assembly code is translated into machine code and placed in memory above the BASIC program text. The machine code program itself is not executed during this process. To execute the program, we must CALL it, as in line 150. Hence assembly (i.e. translation) and program execution are two distinct processes: we assemble the mnemonics into machine code and then, if required, execute that machine code afterwards.

The REPEAT... UNTIL in lines 120 and 170 is a standard way of producing an infinite loop. Notice that we need only assemble the program once: after that we use the technique of passing values to the machine code program using the query operator. It is good programming practice always to design our programs in this way: we should never have to reassemble a program to give it new values to work on.

This last point brings us to the subject of saving programs on tape. Since we should not need to assemble a program more than once into machine code, it is possible in principle to save the machine code on tape, using the instruction *SAVE. Then we need only save lines 120 to 170 of the BASIC program. However it is not generally convenient to store programs in two parts like this. We shall, in general, recommend saving the unassembled program, e.g. all of listing 2.1, and assembling it once each time we load it. This is not likely to be unreasonably slow, and is very much more convenient.

Moreover, this method results in automatically 'relocatable programs' which can be amended in exactly the same way as any other BASIC program. All we need to do is to adjust PAGE prior to loading the program (see Appendix 6). This is a very powerful feature of the BBC computer.

Even if we are producing a stand alone machine code program which we will load with the *LOAD command, we should still keep a copy of the program in assembly language. If we need to add, delete or change a line it then becomes almost as easy as with a BASIC program.

In using an assembler as comprehensive as this one, we do not need to worry very much about what the machine code translation looks like. We can be sure that if our assembly language is correct, the machine code translation will be correct. It follows that if our program fails to work, we should try to mend the assembly language version and not try to delve into the machine code. (We will have more to say about 'debugging' in Chapter 10.) You will have noticed that OPT3 provides us with a listing which includes the machine code translation. This is of some pedogogical interest, and occasionally in this book we will consider the machine code representation of assembly instructions to deepen our understanding. But, in general, our main focus should be at the assembly language level.

## 2.7 STORING NUMBERS LARGER THAN 256 IN ASSEMBLY LANGUAGE

Listing 2.2 illustrates how to load the hex number &D5C3 (decimal 54723) into memory. We need two bytes for this and we use the labels NUML for the low byte and NUMH for the high byte. It is conventional on the 6502 microprocessor to put the low byte in a lower memory location than the high byte and to store them consecutively.

LISTING 2.2

```
   10NUML=&70:NUMH=&71
   20DIM P% 50
   30[OPT3
   40.START
   50LDA #&C3
   60STA NUML
   70LDA #&D5
   80STA NUMH
   90RTS:]
  100CALLSTART
  110?&72=0:?&73=0
  120PRINT!NUML
```

Lines 110 and 120 need some explanation. Line 120 prints out the number represented by the four bytes in &70, &71, &72 and &73, with &73 being the most significant. Since zero is in &72 and &73 (allocated in line 110), the number is &0000D5C3, and line 130 will print out this in decimal. This 'pling' operator (!) is a way of referencing four consecutive memory locations at one go, and we will find it useful later. We could, of course, have printed out the number by writing 256*?NUMH + ?NUML, but listing 2.2 serves to introduce the ! operator.

Finally, there is a useful symbolic convention we can introduce now. NUML and NUMH should really be taken together, since they present one number, and we can usefully write these locations as NUMH;NUML. The semicolon between the two serves to separate two *ordered* bytes, the first being the higher byte and the second the lower byte. We can refer to the *contents* of these locations, and so to the number they jointly represent, as (NUMH;NUML) using the bracket notation we introduced in 2.4. As a decimal number, (NUMH;NUML) is 256*(NUMH) + (NUML). Remember, though, that this notation is to be used for the purposes of *discussion*: it is not part of the 6502 instruction set, and should *not* be used in the assembly programs themselves.

# *Chapter 3 Addition and subtraction*

## 3.1 THE ARITHMETIC UNIT

At the beginning of the last chapter we mentioned that all arithmetic must take place using the accumulator. The part of the microprocessor used to perform the arithmetic is called the Arithmetic Logic Unit (Arithmetic Unit or ALU for short). Figure 3.1 shows the simplified architectural details.

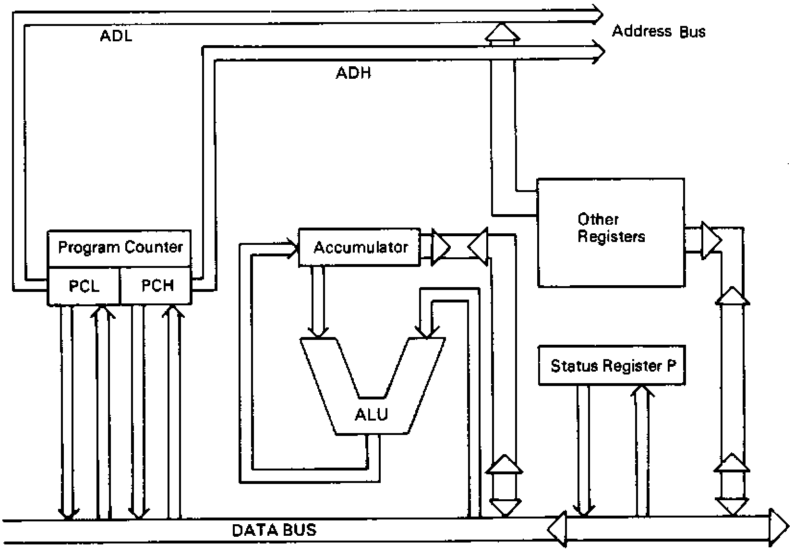


*Figure 3.1 Partial block diagram showing the ALU and Status Register*

Notice that the ALU has two inputs: one from the accumulator and one from the data bus. (In certain internal operations, one of the program counter bytes is used as an input instead of the accumulator, but in order not to complicate things, this is not shown on figure 3.1.) The output from the ALU always passes back to the accumulator (again, with certain internal operations, the output can pass directly to the address bus but this need not concern us now). From the accumulator it can go out on the data bus to a memory location whose address is on the address bus.

Also included on figure 3.1 is a new register, the Processor Status Register, usually labelled P. The function of this register will become clear in this and the next chapter.

## 3.2 WHAT IS THE ASSEMBLY LANGUAGE EQUIVALENT OF SOME SIMPLE BASIC STATEMENTS INVOLVING ADDITION?

(a) Consider first LET SUM = 14 + 29.

As usual, we will assume that SUM has already been given a specific memory location in zero page, for example SUM = &70. Thus we require the result of adding 29 to 14 to be put in memory location &70 which we have labelled SUM.

The assembly coding is:

```
LDA #14

CLC

ADC #29

STA SUM
```

The first statement is familiar by now: put the number 14 into the accumulator. The next one, CLC, means 'clear carry'; we shall come back to the meaning of this later. ADC #29 means add 29 to the contents of the accumulator storing the result in the accumulator. STA SUM stores a copy of this result in SUM.

Notice again the potentially ambiguous use of the label SUM in the BASIC statement. It is very tempting to think of SUM as being equal to 43, whereas it is the *contents* of SUM that is equal to 43 (i.e. (SUM) = 43, using the convention discussed in 2.4). The assembly code version is less ambiguous since the mnemonic STA makes it clear that SUM refers to the *address* of a memory location and not its contents.

(b) Consider now LET SUM = NUM1 + NUM2.

Again, we assume that SUM, NUM1 and NUM2 refer to already specified locations, and that the contents of NUM1 and NUM2 have already been assigned.

The assembly coding is:

```
LDA NUM1

CLC

ADC NUM2

STA SUM
```

Load a copy of the contents of NUM1 into the accumulator, clear carry, add a copy of the contents of NUM2 to the accumulator and store a copy of the result in SUM.

We can represent this process symbolically thus:

$$(\mathrm{NUM1}) + (\mathrm{NUM2}) \rightarrow \mathrm{SUM}$$

Thus the contents of NUM1 and NUM2 are added and stored in the memory location SUM. This symbolic representation is much less ambiguous than the BASIC statement, and we shall be using it a great deal. (Note that some authors reverse this notation i.e. SUM ← (NUM1) + (NUM2).

3.3 THE IMPORTANCE OF CARRY

In 3.2(b) we have implicitly assumed that the result of adding (NUM1) and (NUM2) will not exceed 255 or &FF. This is because the final result will not fit into the one byte SUM otherwise. Does this mean that we are restricted to adding numbers whose sum is less than 256?

Let us consider the situation when (NUM1) = 87 and (NUM2) = 194. The result will be 281, but let us examine the arithmetical process as it occurs in the microprocessor - in binary.

We have:

$$\begin{array}{cccc} & 01010111 & 87 & \\ & 11000010 & 194 & \\ \hline \boxed{1} & 00011001 & 25 & ? \\ \hline \end{array}$$

The result is 25, which is of course wrong. What has happened is that a carry occurred on adding the two most significant bits. Fortunately, however, this carry digit is preserved in one of the bits of the Processor Status Register. This bit is referred to as the *carry flag, C*. Hence, if after performing an addition the carry flag contains 1, we know that a carry has occurred. If no carry has occurred the carry flag will contain zero, regardless of what it contained before the addition took place.

Now 281 in hex is &0119 i.e. 256 + 25. So if there is a carry we need to store our result in two bytes, not one, the higher byte containing 1. How can we access the carry bit however? The answer is that the carry bit is automatically added in when we perform the next addition.

Consider the assembly coding in lines 50 to 120 of listing 3.1.

The first four instructions are as before except that the result is stored in memory location SUML. Then we put zero in the accumulator, store zero in SUMH, and then add SUMH to the accumulator, finally storing the result in SUMH. Why on earth should we want to add zero to zero?

LISTING 3.1

```
 10NUM1=&70:NUM2=&71:SUML=&72:SUMH=&73
 20DIM P% 50
 30[OPT3
 40.START
 50LDA NUM1
 60CLC
 70ADC NUM2
 80STA SUML
 90LDA #0
100STA SUMH
110ADC SUMH
120STA SUMH
130RTS:]
140REPEAT
150INPUT"First number to be added",?NUM1
160INPUT"Second number to be added",?NUM2
170CALLSTART
180PRINT?NUM1+?NUM2,256*?SUMH+?SUML
190UNTIL FALSE
```

The point is that we have done more than this. Look at the mnemonic ADC - what is the C? ADC actually means *add with carry*. So ADC M really means:

$$A + (M) + C \rightarrow A$$

This always happens automatically. It follows that if we do not want to add the carry we should set C to zero. This is the purpose of the instruction CLC.

Notice that the second ADC (ADC SUMH) is not preceded by CLC. This is quite deliberate: if there is a carry from the previous addition we want it included in this addition.

The consequence of this is that if (NUM1) = 87 and (NUM2) = 194, SUML will contain &19 and SUMH will contain 1. Put listing 3.1 into your computer and try some values. Convince yourself that the coding works and then press ESCAPE.

## Exercise 3.1

Write a program to add three numbers together, each number being less than 256. Remember that the final result may be anything up to 765. Type in your program in the same way as in Listing 3.1 and try it out.

## 3.4 ADDING NUMBERS WHICH ARE GREATER THAN 256: MULTIPLE PRECISION ARITHMETIC

How would we program the microprocessor to add two numbers, at least one of which lay between 256 and 65535?

Consider for example 3929 + 52667 or in hex &0F59 + &CDBB. The 6502 microprocessor can only deal with a byte at a time so we consider the lower bytes first. In binary and hex we have:

$$\begin{array}{crr}
 & 01011001 & \&59 \\
 & 10111011 & \&BB \\
\hline
\boxed{1} & 00010100 & \&\boxed{1}14 \\
\hline
\end{array}$$

We have generated a carry of 1, which we carry over to the higher bytes. We then obtain:

$$\begin{array}{rr}
00001111 & \&0F \\
11001101 & \&CD \\
1 & 1 \\
\hline
11011101 & \&DD \\
\hline
\end{array}$$

Thus the final result is &DD14 or 56596, which is correct, of course.

To produce a general coding for this we will put the first number in bytes NUM1H NUM1L, and the second in NUM2H NUM2L. The result is stored in SUMH SUML. Listing 3.2 gives the details. As before put it into your computer and convince yourself that it works. Notice that we cannot print out (NUM1) and (NUM2) by using !NUM1L and !NUM2L, since neither bytes 3 nor 4 of either of these will be zero. We could if we wished output (SUM) by using !SUML as long as &76 and &77 contained zeroes.

LISTING 3.2

```
   10NUM1L=&70:NUM1H=&71:NUM2L=&72:NUM2H=&73:SUML=&74:SUMH=&75
   20DIM P% 50
   30[OPT3
   40.START
   50LDA NUM1L
   60CLC
   70ADC NUM2L
   80STA SUML
   90LDA NUM1H
  100ADC NUM2H
  110STA SUMH
  120RTS:]
  130REPEAT
  140INPUT"First number to be added",!NUM1L
  150INPUT"Second number to be added",!NUM2L
  160CALLSTART
  170PRINT256*?NUM1H+?NUM1L+256*?NUM2H+?NUM2L,256*?SUMH+?SUML
  180UNTIL FALSE
```

You will have noticed that this coding is not quite general enough. It is possible that the result will need three bytes, not two. This will happen if we try to perform 57922 + 37130, for example. However the coding is easily modified for this case, and this is set as an exercise in Exercise 3.2.

Before doing that, let us extend the notation introduced in 2.7 a little. You will recall that we write NUM1H;NUM1L to denote a two byte number. We can extend this to three or more bytes very easily. In this example, if the result is stored in SUM2, SUM1 and SUM0 we can express the whole thing symbolically as:

(NUM1H;NUM1L) + (NUM2H;NUM2L) → SUM2;SUM1;SUM0.

### Exercise 3.2

1. Write a program to perform the operation described symbolically above.

2. Write a program to add together two four byte numbers. Assume that neither number exceeds &7FFFFFFF so that only four bytes are needed for the result.

Set up the addresses using arrays and a loop as follows:

```
10      DIM NUM1(3) , NUM2(3), RESULT(3)
20      FOR I% = 0TO3: NUM1(I%) = &70 + I%
30      NUM2(I%) = &74 + I% :RESULT(I%) = &78 + I% : NEXT I%
```

Test the program by using on INPUT, !NUM1(0) and !NUM2(0). Print out the result by using !RESULT(0).

What is the symbolic representation of this process?

## 3.5 SUBTRACTION

Consider the equivalent assembly code to LET DIFF = NUM1 - NUM2, where both numbers are less than 256 and NUM1 > NUM2.

```
LDA NUM1
SEC
SBC NUM2
STA DIFF
```

The two new instructions are SEC, set the carry flag, and SBC, subtract.

You will be wondering why we need to set the carry flag in this case. In order to understand this we need to understand how the microprocessor subtracts.

In fact the microprocessor doesn't subtract at all - it adds! Thus it treats 98 - 41 as 98 + (-41). The advantage of this is that no additional circuitry is needed for subtraction.

But the problem now is how to represent -41. What we require of the representation for (-41) is that (-41) + 41 = 0. Consider the binary representation of 41: 00101001. Suppose we now reverse all the ones and zeroes to get 11010110 - this is called the *ones complement*, or often just the *complement*. If we add 41 and its complement we get:

$$\begin{array}{r} 00101001 \\ 11010110 \\ \hline 11111111 \\ \hline \end{array}$$

If we now add one to this we obtain:

$$\boxed{1}0000\ 0000$$

Hence, if we ignore the carry, we get zero. It follows that by complementing 41 and adding one, we obtain a representation of -41 i.e. 11010111. This is because it performs just as -41 should perform (-41 + 41 = 0) as long as we ignore the carry obtained.

The process of complementing a number and adding one is called finding the *two's complement* of a number.

Now the microprocessor gets the one it is to add on from the carry flag - this is why we must set the carry flag to 1 before we subtract. What happens if the carry flag is zero will be considered in a moment.

Let us just convince ourselves that this works. In two's complement form 98 + (-41) is:

$$\begin{array}{crr} & 01100010 & 98 \\ & 11010110 & -41 \\ \cline{2-3} \boxed{1} & 00111000 & 57 \\ \cline{2-3} \end{array}$$

and ignoring the carry, this is correct. All that has happened here is that we have performed (57 + 41) + (-41); the last two numbers added together give zero carry 1, and hence we obtain the result.

Notice that the usual understanding of 11010111, namely as the number 215, is replaced by its interpretation as -41. This change in interpretation follows simply because we suppress the usual meaning of the carry flag. If we understand the carry flag in the usual way we would get &139, or 313, which is indeed the sum of 98 and 215. By not treating carry as the 'ninth bit' as we do in addition, we can treat &D7 as -41.

If you have followed all this you will see that when the microprocessor performs the subtraction 215 - 158, it will perform precisely the same sum as above, but in reverse order. Thus 158 is 10011110, its complement is 01100001 and so its two's complement is 01100010. We then get:

$$\begin{array}{cccc} & 11010111 & & 215 \\ & 01100010 & & -158 \\ \cline{2-2}\cline{4-4} \boxed{1} & 00111001 & & 57 \\ \cline{2-2}\cline{4-4} \end{array}$$

as before.

## 3.6 THE FUNCTION OF THE CARRY FLAG IN SUBTRACTION: MULTIPLE PRECISION SUBTRACTION

We have just seen that the microprocessor obtains the one it needs to perform the two's complement from the carry flag. But what happens if zero is in the carry flag?

In this case instead of adding on one it adds on zero. It follows that the number obtained will be one less than it would otherwise have been (since if we add one to a negative number it gets bigger, although it becomes numerically smaller). Thus the effect of having the carry flag zero is to *make the number subtracted one greater*. The *result* of the subtraction will thus be *one less* than would otherwise have been the case.

It is in this way that the carry flag plays a crucial role in multiple precision subtraction. Consider the sum, &732E - &6492. As usual we need to consider the lower bytes first. So we have &2E - &92. The two's complement of &92 is &6E (since &92 + &6E = &100). So in binary we have:

$$\begin{array}{cccc} & 00101110 & & \&2E \\ & 01101110\ + & & -\&92 \\ \cline{2-2}\cline{4-4} \boxed{0} & 10011100 & & \&9C \\ \cline{2-2}\cline{4-4} \end{array}$$

Notice the crucial point: the carry flag is zero. When we worked out 98 - 41 before, it was one. Hence, if *we attempt to subtract a larger byte from a smaller byte the carry flag is reset to zero*.

What do we make of the final result &9C? There are two ways to look at it: one is to treat it as a negative number (we shall return to this in a moment). The other way is to consider it as the result we get if we have borrowed one from the next column.

Thus:

$$\begin{array}{r} \&12E \\ 92\ - \\ \hline \&\ 9C \\ \hline \end{array}$$

(remember &12 = 18 and 18 - &9 = &9).

This second way of treating things is what is required here. The carry being reset to zero indicates that we have needed to borrow. Now as we have seen, if the carry is zero this is equivalent to subtracting a number one greater. Hence when we subtract the higher bytes we will 'pay the borrow back'. Thus, the one's complement of &64 is &9B (&64 + &9B = &FF), and this is the two's complement of &65. We have then:

$$\begin{array}{rlcl} & \&73 & \qquad & \&73 \\ & \&9B\ + & & \&65\ - \\ \cline{2-2}\cline{4-4} \boxed{1} & \&0E & & \&0E \\ \cline{2-2}\cline{4-4} \end{array}$$

The final result is &0E9C, which is correct, of course.

Now, can we always be sure that when we try to take a larger byte from a smaller one the carry flag will be reset to zero? Indeed so. The two's complement of a number is 256 minus that number. Hence V - U gives V + (256 - U). Now if U > V this must be less than 256 since we can write it 256 - (U - V). Similarly if V > U it must be more than 256 since we can write it 256 + (V - U). It always works.

Exercise 3.3

1. Write a program to perform two-byte multiple precision subtraction as described above.

The symbolic representation is:

(NUM1H;NUM1L) - (NUM2H;NUM2L) → DIFFH;DIFFL

2. Using the same labels as in Q.2, Ex.3.2 write a program to subtract two four byte numbers.

## 3.7 POSITIVE AND NEGATIVE NUMBERS: SIGNED ARITHMETIC

We have seen that any byte can be considered as negative or positive: the interpretation we (or the microprocessor) give to it depends on how we obtained the number, and the various conditions applying at the time.

Above we said that in &2E - &92 = &9C, we could consider &9C as a negative number. In that case it is -&63 or -99. But we would have got &9C as an answer to &ED - &51, and this time it would have been positive i.e. +157. We know which is which because in the first case the carry flag is reset to zero, and in the second case it is set to one.

Now suppose that, instead of obtaining a negative number as a result of a subtraction sum, we just wanted to input a negative number straight away. In principle there seems to be no problem: if we want -17 we need the two's complement of &11 which is &EF and we can use this; similarly if we want -150 we use &6A. As long as we remember that in both cases the numbers are to be understood as negative then there is no problem, it seems. Unfortunately, this is totally impracticable. Almost all the time, we will want to do arithmetic with such numbers, and it would defeat the object of using a microprocessor if we had to work through all the sums first so that we could tell the microprocessor whether a result is to be understood as positive or negative! Thus, for example, the micro may at one point perform the sum &6A + &12 = &7C. Is this 124 or -132? The microprocessor has no clues, since it has taken the byte &6A from memory and there is no room for us to attach a sticky label to the byte indicating that it is negative (if that's what we require it to be)!

We are going to have to introduce a convention to deal with this. If a number is going to be considered negative then its most significant bit must be one; otherwise it is a positive number. This means that using a single byte to represent a number, &00 to &7F are positive; and &80 to &FF are negative. This gives us a range of -128 to +127. This convention is an easy one to use since if a number is greater than 127 we need to subtract it from 256 and treat the result as negative e.g. &AB is -&55 i.e. -85. It is usual to call the most significant bit in this case the *sign bit*.

Remember that this convention is only necessary if we want to treat some numbers as negative *from the outset*; in all other cases, there is other information in the microprocessor (like the state of the carry flag) to provide the clue as to the sign of the result.

Let us consider a couple of examples:

```
(a) 70 + (-100)          01000110
                         10100110 +
                         --------
                         11101100
                         --------
```

Since the MSB is a 1, the number is negative, viz -30.

(b) (-24) + (-58)

$$\begin{array}{rl} & 11101000 \\ & 11000110\ + \\ \cline{2-2} \boxed{1}. & 10101110 \\ \cline{2-2} \end{array}$$

Again the result is negative, viz -82. The carry flag here has no significance. This is because we are effectively only dealing with seven bits; any carry will go into the eighth bit, though even then it may not always be relevant.

This last point leads us to a problem called overflow. Consider

(c) 83 + 54

$$\begin{array}{l} 01010011 \\ 00110110\ + \\ \hline 10001001 \\ \hline \end{array}$$

The result is -119, which is plainly ridiculous. Moreover, the microprocessor knows it is ridiculous too, since it has added two positive numbers. It therefore flags the problem by setting another bit in the status register called the overflow bit, and labelled V (oVerflow).

It should be obvious that overflow can only occur when adding two positive or two negative numbers i.e. when adding two numbers of the same sign. In those cases, and *only* in those cases, can V be set to one. Overflow is often erroneously thought of as a carry taking place from bit 6 to bit 7, but a glance at example (b) will show you this is *not* so.

The overflow flag is set to one by the microprocessor only if the sign bit of the result is different from the sign bits of the two numbers added (assuming, of course, that the numbers have the same sign bit); in all other cases it is reset to zero. Overflow can occur without there being any carry from bit 6 to bit 7 (consider for example (-65) + (-65)).

What happens if V is set to 1? Then we know the result is either greater than 127 or less than -128. Which it is will depend on the value of the MSB: if it is one, the result is greater than 127; if it is zero, less than -128. We will consider how to deal with these cases in the next chapter.

Let us end this section by considering how we can represent signed numbers in two bytes. Using two bytes we have &0000 to &7FFF as positive (MSB = 0), and &8000 to &FFFF as negative (MSB = 1). This gives us a range of $-2^{15}$ to $2^{15} - 1$ i.e. from -32768 to +32767. Let us consider as an example 18548 + (-25239).

In hex we have:

```
&48      &74
&9D +    &69 +
───      ───
&E5      &DD
```

The result is &E5DD, which is negative since the MSB = 1 (E = 1110), and this is -6691. A simple way to calculate this is -(255 - &E5) * 256 - (256 - &DD) = -6691, or -(&10000-&E500).

Notice that the arithmetic on the low bytes is interpreted quite normally. The V flag *will* be set, but *we* choose to take no notice of it because the sign bit is not bit 7 but bit 15. The microprocessor will always set V when treating the bytes as single byte signed numbers would result in overflow: it doesn't know whether we are treating them this way or not in any particular case. In this case it is the higher pair of bytes (&48 and &9D) that are being considered this way, but in this case overflow doesn't occur. Remember, then, that overflow is only a problem to *us* when we are considering the bytes which contain the sign bit. The microprocessor will always flag overflow when appropriate in case the particular pair of bytes we have just added *do* contain the sign bit: it is up to us to make the right interpretation in the way we construct our programs, as we will see in the next chapter.

## Exercise 3.4

1. Show how the computer would perform the sums 24 - (-18) and (-86) - 35. Under what circumstances would overflow occur in the case of signed subtraction?

2. What is the range of signed numbers with 4 bytes?

3. What adjustments, if any, would you need to make to the program of Q.2 in Exercises 3.2 and 3.3 to perform signed addition and subtraction, assuming there is no overflow? Give a BASIC statement to print out the result from the four bytes.

## 3.8 LOGICAL OPERATIONS

Suppose we have a number in the accumulator and we wish to change its sign i.e. we want to perform the operation -A → A. One way to do this is to subtract the accumulator from zero, but to do this we need to put the accumulator into a memory location. The coding is:

```
STA TEMP
LDA #0
SEC
SBC TEMP
```

where TEMP is the temporary memory location.

It would be helpful to have a way of doing this which does not need any temporary memory location; and indeed there is. We need a new operation called the exclusive-OR. Actually this operation exists in BASIC and indeed has the same mnemonic, EOR.

EOR operates only on the accumulator and the final result is put in the accumulator. EOR M takes each bit of the accumulator and exclusive-ORs it with the corresponding bit of (M), according to the following table:

| ⊻ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

where ⊻ is the symbol for EOR. Like bits give a zero, unlike bits a 1.

It follows that to complement the accumulator we need only write EOR #&FF: every zero in the accumulator becomes 1 and every 1 becomes zero. To give the two's complement we need to add 1. So the full coding to perform -A → A is:

```
EOR #&FF
CLC
ADC #1
```

For completion now let us mention the other two logical operations the microprocessor can perform, both on the accumulator (and both also present as BASIC operations).

First ORA M. This takes each bit of the accumulator and ORs it with the corresponding bit of (M), according to the following table:

| V | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

where V is the symbol for OR. The final result is put in the accumulator. Only two zeroes give a zero, all the other combinations give one.

The main use of ORA is to set one or more bits to one in the accumulator without affecting the rest. For example, ORA #&42 will ensure that bits 1 and 6 are both 1, and that the others are unchanged.

The final operation is AND M. This takes each bit of the accumulator and ANDs it with the corresponding bit of (M), according to the following table:

| Λ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

where Λ is the symbol for AND. The final result is put in the accumulator. Only two ones give a 1, all the other combinations give zero.

The main use of AND is to *mask* off one or more bits in the accumulator, i.e. to set them to zero, without affecting any other bits. For example, AND #&ED will ensure that bits 1 and 4 are zero, and that the other bits are unchanged. AND #&0F will ensure the top four bits are zero, the bottom four remaining unchanged.

The uses of these operations will become more apparent in the later chapters of this book.

### Exercise 3.5

Write the coding to:

(a) set bits 3 and 7 to one and to mask bits 0 and 4, all other bits remaining unchanged;

(b) reverse bit 7, set bit 6 to 1 and mask off all the rest of the bits except bit 5.

## 3.9 A NEW ADDRESSING MODE: IMPLIED ADDRESSING

The instructions CLC and SEC illustrate a new form of addressing. All instructions must refer to some part of memory or to some register, and this includes the instructions CLC and SEC. In this case, it is the carry flag in the Processor Status Register that is addressed and the value to be deposited in that flag is included in the instruction. This is implicitly the case, however, since we do not need to specify what is being addressed nor what value is to be deposited, aside from writing down the instruction itself. Instructions like these, where the address and contents referred to are included implicitly in the instruction, are said to use the *implied addressing mode*. All such instructions are single byte, for no data is required to accompany the instruction. Note that instructions like EOR are not implied: for whilst one of the addresses is implicit - the accumulator - a second one is needed, namely the memory location with whose contents the accumulator will be EORed. Everything must be included in the instruction for it to be implied addressing.

# *Chapter 4 Decision-making in assembly language*

## 4.1 THE PROCESSOR STATUS REGISTER

In the last chapter (Section 3.1) reference was made to a new register: P, the status register. Figure 3.1 shows it in relation to the rest of the architecture we have met so far. Each of the eight bits of this register has a separate function: each operate as *flags,* indicating that certain conditions have or have not occurred. If the bit is 1, the flag is said to be *set,* if the bit is 0, the flag is said to be *clear* (or sometimes *reset*). We have already considered two flags: C, the carry flag, and V, the overflow flag. Figure 4.1 shows all the flags in the P register. There are five new ones, N, B, D, I, Z. Of these, we will defer discussion of B, D and I until later chapters.

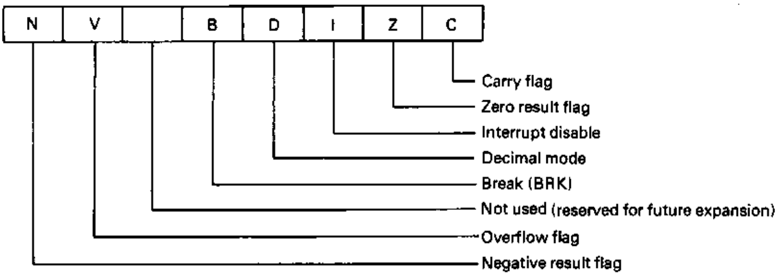


*Figure 4.1: Processor Status Register*

The N flag is set to one whenever the most significant bit of the result of any arithmetic or logical operation is one. It is also set to one if data moved into the accumulator has an MSB of one. (It is also affected by movements concerning the X or Y registers, as we shall see in Chapter 5.) It is called the negative flag because if a byte is to be interpreted as a signed integer, then a negative integer will have an MSB of one. This concept of negativity has no meaning outside of the signed integer representation, but the negative flag has uses outside signed arithmetic. Since N = 1 when the MSB of a result is one, and N = 0 when the MSB of a result is zero, the N flag allows us easily to test whether the MSB of a location is set or not. This has particular uses in monitoring input from an external device. In this context, it is often used with the BIT instruction, which we will consider in Chapter 9.

The Z flag is set to one whenever the result of any arithmetic or logical operation is zero. It is also set to one if data moved into the accumulator is zero (and if data moved into the X or Y registers is zero, as we shall see in Chapter 5). Thus the Z flag is affected by exactly the same operations as the N flag, but it reflects a result of zero rather than an MSB of one.

One particular use of the N, V, Z and C flags is in the monitoring of branching within the microprocessor, and this is the concern of this chapter.

## 4.2 DECISION-MAKING USING THE MICROPROCESSOR

You will know from your work with BASIC that the real power of a computer lies in its ability to take simple decisions. This is the conditional IF...THEN jump in BASIC - what corresponds to it in assembly language? The general concept involved is one of *branching*.

Usually, the program counter moves consecutively through a portion of memory. However, there are a series of instructions which cause it to stop doing this and instead to jump forward or backwards by a certain amount if a particular flag is set or clear. These instructions are called the branch instructions, and all their mnemonics have a first letter of B, for 'branch'. Here they are:

| | | | |
|---|---|---|---|
| BEQ | branch if | Z = 1 | (result EQual to zero) |
| BNE | branch if | Z = 0 | (result Not Equal to zero) |
| BMI | branch if | N = 1 | (data MInus; MSB = 1) |
| BPL | branch if | N = 0 | (data PLus; MSB = 0) |
| BCS | branch if | C = 1 | (Carry Set) |
| BCC | branch if | C = 0 | (Carry Clear) |
| BVS | branch if | V = 1 | (oVerflow Set) |
| BVC | branch if | V = 0 | (oVerflow Clear) |

These instructions are all *conditional*: the branch only occurs if the condition is true (i.e. if the relevant flag has the state indicated). You will recall the GOTO statement in BASIC, which is *unconditional*, since the jump is always made. A similar instruction exists in assembly language:

```
JMP     jump to a new location in memory and commence
        program execution from there
```

The program counter has its contents replaced by a new address, and this in effect, causes a jump to a new section of the program.

Clearly all the above instructions need an operand: they all need to have attached a reference which points to the location where the branch or jump should go. In BASIC this is done by using line numbers: in assembly language, the corresponding concept is the *label*. We have already met this idea in Chapter 2 with .START: this label marked the beginning of the program since it was put next to the very first instruction. You will recall that a label can be any BASIC variable, but that it must begin with a dot. It is very useful to make these labels as meaningful as possible (.START is better than .X2Y96T, though both would serve the same purpose).

The best way to illustrate the concept of branching and of labels is to see how we can translate simple BASIC statements into assembly language.

## 4.3 THE ASSEMBLY LANGUAGE EQUIVALENTS OF SOME BASIC CONDITIONAL STATEMENTS. I: USE OF THE N AND Z FLAGS

(a) IF NUM = 0 THEN GOTO 50

```
                 LDA NUM
                 BEQ ZERO
                        .
                        .
                        .
                        .
         .ZERO          .
                        .
                        .
                        .
```

There are a number of points to make here. Notice first the way the label, ZERO, is put in the first column; the second column contains the instruction and the third the operand. This is a standard way to write down assembly language programs. Notice next that the dot in front of the label is only necessary if the label is put in the first column; when ZERO is an operand (as it is in BEQ ZERO) the dot is not required. As we will see later, the dot is used by the BBC microcomputer to differentiate labels from instructions; it is not a standard requirement of assemblers. Indeed, very few assemblers use this convention. For this reason, unless we are displaying a program which is to be typed directly into the microcomputer, we will not include the dot notation either. We shall have more to say about this in Section 4.6.

Notice the choice of label here: ZERO is a meaningful choice since the branch is only made if the result is zero. Again, any legal label will do, but we should try to find ones which indicate to us what is going on. The program works as follows: we load the contents of the location with address NUM into the accumulator. BEQ ZERO looks at the Z flag and branches to the line with the label ZERO if Z = 1. If Z = 0, the branch is not made and the next instruction is obeyed as usual. Thus this assembly program does exactly the same as the BASIC statement.

In the program shown, the line labelled ZERO is further down in memory than the branching instruction, but this is not essential. Branches can occur backwards as well as forwards. In that case the program would be:

```
ZERO        .
            .
            .
            .
            .
         LDA NUM
         BEQ ZERO
```

(b) IF NUM <> 0 THEN GOTO 50

```
         LDA NUM
         BNE NTZERO
             .
             .
             .
             .
NTZERO       .
```

Here a branch is made to the line labelled NTZERO if Z = 0 (i.e. if (NUM) ≠ 0). Again the labelled line could be earlier rather than later than the branching instruction: this is true of all the branching instructions, in fact. Notice again the use of the meaningful label. Remember that in some assemblers labels are confined to a length of six symbols, but the BBC micro allows us to use labels of whatever length we like.

(c) IF NUM < 0 THEN GOTO 50

```
           LDA NUM
           BMI NEGATIVE
               .
               .
               .
               .
NEGATIVE       .
```

This assumes, of course, that we are working with the signed integer representation . otherwise it would not make sense to test for a number to be less than zero. This and the next two examples are valid only if numbers are in signed integer format.

(d) IF NUM >= 0 THEN GOTO 50

```
                      LDA NUM
                      BPL GTEQUAL
                           .
                           .
                           .
                           .
             GTEQUAL       .
```

(e) IF NUM > 0 THEN GOTO 50

```
                      LDA NUM
                      BEQ ZERO
                      BPL POSITIVE
                ZERO       .
                           .
                           .
                           .
                           .
            POSITIVE       .
```

In (d) we note that if a number is zero or positive, then N = 0, and so BPL will result in a branch if (NUM) is greater than or equal to zero. If we want the branch to occur only if (NUM) is greater than zero, as here, we must first eliminate the case when (NUM) equals zero. This is the function of BEQ ZERO. If we get through to BPL POSITIVE then either (NUM) is more than zero or less than zero. If the former, then the branch to the line labelled POSITIVE will occur; otherwise the next instruction will be performed and this, of course, is the one labelled ZERO. Hence the set of instructions beginning with the line labelled POSITIVE are performed if (NUM) > 0; in all other cases, the set of instructions beginning with the line labelled ZERO are performed.

(f) IF NUM = 0 THEN NUM = 1

```
                      LDA NUM
                      BNE NTZERO
                      LDA #1
                      STA NUM
              NTZERO  ———
```

Many IF... THEN statements in BASIC do not involve jumps at all In assembly language, however, conditional statements must always be effected by a branch, as here. The most economical way to encode this is to branch to what is in effect the next statement (at NTZERO) if (NUM) ≠ 0. Then, if no branch occurs (NUM) must be zero, and then the second part of the conditional statement is enacted. Here is another example.

(g) IF NUM < 0 THEN NUM = -NUM

```
            LDA NUM
            BPL NTMINUS
            EOR #&FF
            CLC
            ADC #1
            STA NUM
NTMINUS
```

(h) IF NUM = 0 THEN NUM1 = 3: NUM2 = 4 ELSE NUM1 = 4: NUM2 = 3

```
            LDA NUM
            BNE NTZERO
            LDA #3
            STA NUM1
            LDA #4
            STA NUM2
            JMP OVER
  NTZERO    LDA #4
            STA NUM1
            LDA #3
            STA NUM2
    OVER    ————
```

An IF... THEN... ELSE structure in BASIC can only be accommodated in assembly language by using an unconditional jump, for this separates the THEN... from the ELSE... . Some authorities on assembly language dislike the use of the JMP instruction and prefer if possible to replace it by a branch instruction that always branches. In the example here BNE OVER would do that, since the accumulator always contains &04 which is non-zero. They have two reasons for doing this: firstly, as we shall see later in this chapter, the branch saves a byte of storage (though it executes no quicker); secondly, as we shall also see later, the branch is intrinsically relocatable, in that its addressing mode is relative, whereas the jump uses an absolute address.

These points are valid, but must be offset against the unnaturalness of the coding. Using such 'tricks' as these needs very careful documentation, otherwise disasters occur. If we use the coding as a model and then try to implement IF NUM = 0 THEN NUM1 = 3: NUM2 = 0..... , the branch will not work.

Exercise 4.1

Encode these BASIC statements using assembly language with suitable labels (in all the questions (NUM1) and (NUM2) occupy single byte locations - in 1 assume signed integer representation).

1. IF NUM1 + NUM2 > 0 THEN GOTO 50.
2. IF NUM1 - NUM2 <> 0 THEN NUM3 = NUM2: NUM2 = NUM1: NUM1 = NUM3.
3. IF NUM1 + NUM2 > 255 THEN SUM = 0 ELSE SUM = NUM1 + NUM2.

## 4.4 THE ASSEMBLY LANGUAGE EQUIVALENTS OF SOME BASIC CONDITIONAL STATEMENTS. II: USE OF CMP INSTRUCTION

(a) IF NUM1 = 2 THEN GOTO 50

Here is the way to encode this using the instructions learnt so far:

```
              LDA NUM1
              SEC
              SBC #2
              BEQ EQUAL
                   .
                   .
                   .
                   .
        EQUAL      .
```

This coding is fine if we do actually want to reduce (NUM1) by 2. But if our sole aim is to *compare* (NUM1) to 2 then putting the result of subtracting 2 into the accumulator, as we have done here, is a waste of time. What we require is an instruction which will do the subtraction and set the appropriate flags *without* affecting the accumulator. This is the function of the CMP instruction. The coding using CMP is:

```
              LDA NUM1
              CMP #2
              BEQ EQUAL
                   .
                   .
                   .
                   .
        EQUAL      .
```

This is much neater and leaves (NUM1), unaffected, in the accumulator. Notice that we do not need SEC here: CMP will set carry automatically before doing the subtraction.

(b) IF NUM1 >= 6 THEN GOTO 50

Without the CMP instruction we would need to subtract 6 and see if the result were greater than or equal to zero. Using CMP, we can do the subtraction and affect the appropriate flags without having the subtraction contaminate the accumulator. We have:

```
                 LDA NUM1
                 CMP #6
                 BCS GTEQUAL
                      .
                      .
                      .
                      .
        GTEQUAL       .
```

CMP #6 subtracts 6 from the accumulator, sets the relevant flags and then *throws away the result of the subtraction*: the contents of NUM1 are still in the accumulator, quite unaffected.

You may be wondering why we have not used BPL GTEQUAL here as we did in (d) of Section 4.3. To understand this, suppose that (NUM1) = 135. We take away 6 and get 129, which is &81. The MSB is one and therefore the result is negative! BPL can only be used to test if bit 7 of some data is zero; in a few cases, like 4.3(d), this can be interpreted as testing if a result is positive, but in most cases it cannot be used like this. Generally speaking, *do not use BPL or BMI with the result of a CMP instruction: use BCS and BCC instead.*

To understand why BCS works here, recall what happens to the carry flag when we subtract. If no borrow has occurred, C will equal one (i.e. carry will be set); if a borrow has occurred C will equal zero (i.e. carry will be clear). In the former case, the number subtracted cannot be greater than the number it is subtracted from. So we have:

| | |
|---|---|
| BCS | Branch if result greater than or equal to zero |
| BCC | Branch if result less than zero |

This is not all that easy to remember, and the particular mnemonic I use for BCC is 'Branch if result of Comparison a Complement'. This reminds me that BCC only branches if the result is negative; by extension, BCS branches in all other cases.

(c) IF NUM < 35 THEN GOTO 50

```
                    LDA NUM
                    CMP #35
                    BCC LESS
                          .
                          .
                          .
                          .
            LESS          .
```

Here the branch occurs if (NUM) - 35 is negative i.e. if NUM < 35. Equality will not give a branch.

(d) IF NUM >= -6 THEN GOTO 50

```
                LDA NUM
                BPL GTEQUAL
                CMP #&FA
                BCS GTEQUAL  (or BPL GTEQUAL)
                      .
                      .
                      .
                      .
       GTEQUAL        .
```

We suppose here, of course, that NUM is to be interpreted as a signed byte. We check first if NUM is positive or zero with BPL. If not, then we make the comparison with -6, which is encoded of course as &FA (the BBC assembler will not allow CMP #-6). Hence &FF to &FA will give the branch (the result is 5 to 0) and &F9 to &80 will not (these give -1 to -122).

Notice that

```
                LDA NUM
                CMP &FA
                BPL GTEQUAL
```

will not work. Consider for example NUM = 122, which results in &80 or -1.

(e) IF NUM1 > NUM2 THEN GOTO 50

```
                LDA NUM1
                CMP NUM2
                BEQ EQUAL
                BCS GREATER
                    .
        EQUAL       .
                    .
                    .
                    .
                    .
       GREATER
```

The logic of this is similar to 4.3(e). Notice that we are not restricted to the immediate addressing mode with CMP (although it is often used like this in practice). In fact, CMP uses most of the addressing modes available (of which we have so far met Absolute and Zero Page - clearly Implied is not applicable). In the case here, the contents of NUM2 are subtracted from the accumulator, the result being thrown away.

It may help to summarise the cases follow a CMP.

| *Branch if* | *Instruction* | *Suggested label* | (followed by: *Instruction* | *label*) |
|---|---|---|---|---|
| Acc < memory | BCC | LESS | | |
| Acc ⩽ memory | BEQ | LESSEQ ; | BCC | LESSEQ |
| Acc = memory | BEQ | EQUAL | | |
| Acc ≠ memory | BNE | NTEQUAL | | |
| Acc ⩾ memory | BCS | GTEQUAL | | |
| Acc > memory | BEQ | EQUAL ; | BCS | GREATER |

## Exercise 4.2

Encode these BASIC statements into assembly language.

1. IF NUM > 15 THEN NUM = 0.

2. IF NUM <= -10 THEN INDIC = NUM ELSE NUM = 0: INDIC = 1.

3. IF NUM1 = NUM2 THEN NUM2 = NUM3 ELSE IF NUM4 > 16 THEN NUM2 = NUM4 ELSE NUM2 = 0: NUM4 = 0.

## 4.5 COMPARING NUMBERS GREATER THAN 255

Consider first how we can test if (NUM1H; NUM1L) = (NUM2H; NUM2L). Here is the assembly code:

```
                    LDA NUM1L
                    CMP NUM2L
                    BNE NTEQUAL
                    LDA NUM1H
                    CMP NUM2H
                    BEQ EQUAL
          NTEQUAL        .
                         .
                         .
                         .
                         .
                         .
                         .
                         .
            EQUAL        .
```

The idea here is to establish first whether the lower bytes are equal: if they're not, we can jump straight to NTEQUAL. If they are, we compare the higher bytes, and if they're equal, then both two byte numbers are equal.

Consider now how we test for (NUM1H; NUM1L) ⩾ (NUM2H; NUM2L). The easiest way to do this is to perform a subtraction and see if the result is non-negative. The coding for this is straightforward.

```
                    LDA NUM1L
                    SEC
                    SBC NUM2L
                    LDA NUM1H
                    SBC NUM2H
                    BCS GTEQUAL
                         .
                         .
                         .
                         .
          GTEQUAL        .
```

The idea here is that if no borrow is required when subtracting the high bytes (i.e. if C = 1 after this subtraction) then the first two byte number must be at least as large as the second one. Notice that any borrow required from subtracting the low bytes is 'paid back' automatically in the second SBC instruction.

Now, an inspection of this code points to a way of shortening it. All we use from the first three lines is the carry flag: if set to one, there is no borrow, if zero, there is a borrow. We are not interested in the result of the subtraction. So we can replace the first three lines by two, involving CMP as follows:

```
                LDA NUM1L
                CMP NUM2L
                LDA NUM1H
                SBC NUM2H
                BCS GTEQUAL
                     .
                     .
                     .
                     .
        GTEQUAL      .
```

The carry flag is affected in just the same way, but we save a line since we do not need to set carry initially with SEC when using CMP.

Consider finally how to test for (NUM1H; NUM1L) > (NUM2H; NUM2L). This may seem identical to the last case, but a little reflection will show you why it is not. We have no way in the last piece of coding of deciding whether the numbers are indeed equal. It does not suffice to know that the high byte difference is zero, since this would occur with, for example, &1A86 - &1A52 and with &5C49 - &5BA3.

Can we in some way combine the first two tests economically? Clearly if we compare the lower two bytes and they are not equal then we cannot have equality. So let us focus on the case when the lower bytes are equal. Consider (a) &149A - &139A and (b) &149A - &149A. Now in (a), if we pretended that there were a borrow from the lower byte subtraction, then the result of the higher byte subtraction would still be non-negative, whereas in (b) it would be negative. This provides the clue: if the lower bytes are equal we shall engineer a borrow before subtracting the higher bytes. Here is the coding:

```
                LDA NUM1L
                CMP NUM2L
                BNE NTEQUAL
                CLC
        NTEQUAL LDA NUM1H
                SBC NUM2H
                BCS GREATER
                     .
                     .
                     .
                     .
        GREATER      .
```

CLC at line 4 engineers the borrow we require.

Exercise 4.3

Encode into assembly language, tests for:

1. (NUM1H; NUM1L) < (NUM2H; NUM2L).

2. (NUM1H; NUM1L) ≤ (NUM2H; NUM2L).

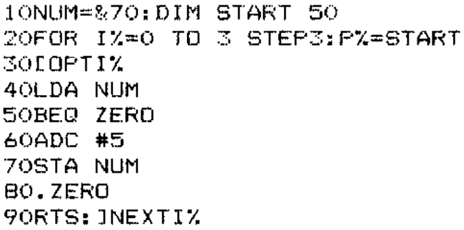
3. (NUM1(3); NUM1(2); NUM1(1); NUM1(0)) ≥ (NUM2(3); NUM2(2); NUM2(1); NUM2(0)).

4. Can you find an economical way to test if (NUM(3); NUM(2); NUM(1); NUM(0)) = 0? (Hint: Use ORA.)

## 4.6 TYPING IN ASSEMBLY LANGUAGE PROGRAMS WITH LABELS INTO THE BBC MICROCOMPUTER

Listing 4.1 illustrates how to put labelled assembly programs into the BBC microcomputer. There are a number of new features here. First notice that the whole program from line 30 to 90 is assembled twice. On the first *pass*, as it is usually called, we use OPT0: this suppresses both listing and errors. On the second pass, we use OPT3 which gives both the listing and the errors. The reason we suppress the errors on the first pass is that an error is bound to happen. When the assembler gets to line 50, it does not know what value to give to the label ZERO because it hasn't met it yet; ZERO doesn't appear until line 80. Therefore, it treats this as an undefined label error and leaves it unassigned. On the second pass, however, it has already met ZERO (in line 80 on the first pass) and can therefore assign a value to it. This two-pass idea must always be used if there are forward references in a program. A forward reference is the use of the label as an operand in a branch or jump instruction which has not yet been met: that is, the branch or jump is to a higher line number.

LISTING 4.1

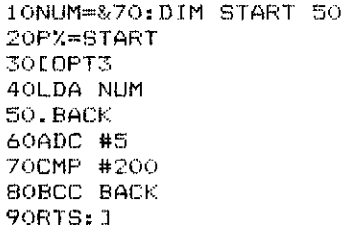


```
10NUM=&70:DIM START 50
20FOR I%=0 TO 3 STEP3:P%=START
30[OPTI%
40LDA NUM
50BEQ ZERO
60ADC #5
70STA NUM
80.ZERO
90RTS:]NEXTI%
```

When we have to make two passes we need to change the way we allocate memory using the DIM statement. We cannot use DIM P% as before, because on the second pass a new set of memory locations will be allocated. What we require to happen is that the translation from the second pass goes into the same locations as those from the first pass. If this does not happen, the values of the labels from the first pass will not apply to the second. To ensure this, we give the address of the first location to the variable START using DIM START. Now on both passes, P% = START will put the start of the machine code in the same place. This method also dispenses with the need to use a label .START to mark the beginning of the program.

Listing 4.2 shows that two passes are not always required when using branches and jumps. Here, the branch at line 80 is defined on the first pass since BACK is defined in line 50. However, in more complex programs involving labels it is sometimes tedious to ascertain whether or not forward references are made. It does no harm, therefore, to use the two-pass method in listing 4.1 if in doubt, even if it turns out that two-passes weren't in fact necessary. You can discover if two passes are necessary by using FOR I% = 1 TO 3 STEP 2 and seeing if all labels are assigned on the first pass (OPT1 gives a listing but suppresses errors).

LISTING 4.2



```
10NUM=&70:DIM START 50
20P%=START
30[OPT3
40LDA NUM
50.BACK
60ADC #5
70CMP #200
80BCC BACK
90RTS:]
```

Notice that in both listing 4.1 and 4.2 the coding is not arranged in three columns. Thus if a label is used it is put in a line on its own, as in line 80 of 4.1. This is why a dot is required for column one labels on the BBC computer, for otherwise the assembler could not differentiate between a label and a mnemonic. It is possible to put the label on the same line as the instruction (e.g. .NOOVFLOW BPL NTNEG) but if we do this the column structure is destroyed and readability greatly impaired. We could preserve columns by indenting all the other lines by 10 spaces, but this would be exceptionally tedious. Hence, putting column one labels on single lines is the best solution, since it preserves columns two and three. This method is unique to the BBC assembler: other assemblers will allow simple formatting into three columns. But the advantage with the BBC computer is that labels are of unrestricted length, a feature missing from most assemblers. Programs on paper should still be written in three columns, but when typed into the computer the two column format with column one labels on a single line should be used.

One final point. It is possible to use multi-line statements if required. For example, we could compress listing 4.1 into listing 4.3 if we liked. There are two problems with this: firstly, editing is more difficult; and secondly, the program is harder to read since the column structure is lost. It is a matter of preference, but in this book listings will use single line statements most of the time.

LISTING 4.3

```
10NUM=&70:DIM START 50
20FOR I%=0 TO 3 STEP 3:P%=START
30[OPTI%:LDA NUM:BEQ ZERO:ADC #5:STA NUM:
   .ZERO RTS:]NEXTI%
```

## 4.7 RELATIVE ADDRESSING

Look again at the assembly listing for Listing 4.1, and focus in particular on the machine code translation of line 50. It is F0 04. F0 is the opcode for BEQ, but what does the operand, 04, mean?

The mode of addressing used here is called *relative addressing*. The branch is made to an instruction located 4 bytes further on from that pointed to by the present contents of the program counter. That is, if Z = 1, four is added to the contents of the program counter, and this achieves the branch. Figure 4.2 shows this in more detail. The top location represents the beginning of the program, whose address is &0E93 the value of START, and so the PC is set initially to &0E93. The microprocessor thus loads the accumulator with the contents of location &70 and the PC is now set at &0E95. Now the next byte, F0, is fetched and decoded as a BEQ instruction; meanwhile the next byte is fetched, 04, and this is understood as the displacement if the branch occurs. At this stage the PC is at &0E97, the address of the next instruction (ADC). Hence, if the branch occurs, the displacement is added to the address of the next instruction. This is quite complicated to work out, and if we had to do it mistakes would be common. Fortunately, the assembler allows us to use labels and it does the calculations for us.

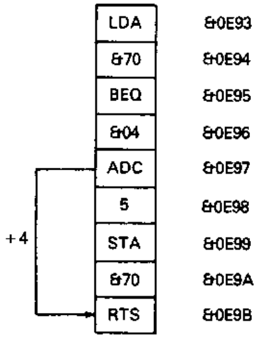


*Figure 4.2: A forward branch*

Look now at the assembly listing for Listing 4.2, and focus on line 70. The translation is 90 FA. 90 is the opcode for BCC, and FA is interpreted as a signed integer i.e. as -6. So if the branch occurs (i.e. if C = 0), 6 is taken from the contents of the PC. Figure 4.3 shows the details.

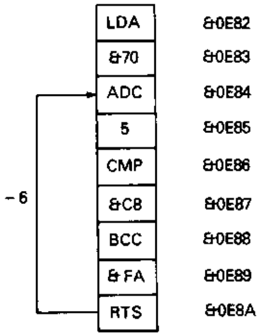


*Figure 4.3: A backward branch*

Hence the displacement to be added to the contents of the PC is always understood as a signed integer. This means that the maximum forward branch that can be made is +127 from the instruction after the branch i.e. +129 from the branch instruction itself. Similarly, the maximum backward branch is -128 from the instruction after the branch i.e. -126 from the branch instruction itself. Figure 4.4 shows this diagrammatically.

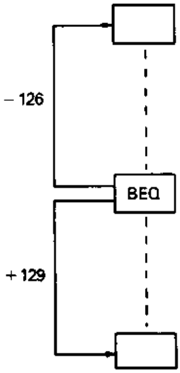


*Figure 4.4: The maximum range for a branch*

If we try to make a branch beyond this, the assembler will give us a 'branch out of range' error. Now most of our branches will be within this range, but what happens if we wish to go beyond it? Suppose that in line 50 of listing 4.1 the label ZERO refers to an address more than 129 from &0E95, the address of the BEQ instruction. We can recode it thus:

```
                  BNE NOJUMP
                  JMP ZERO
         NOJUMP
```

Similarly in line 70 of listing 4.2, if the branch is out of range we can recode

```
                  BCS NOJUMP
                  JMP BACK
         NOJUMP
```

Hence we solve the problem by *not* branching if the condition we require holds and then jumping to the desired place.

This solution is not ideal since it uses 5 bytes of storage instead of two, but it should not be required too often. We could in fact try to do it in four bytes by having as the original branch, BEQ PIVOT in listing 4.1, where PIVOT is an intermediate point where a BEQ ZERO is located, ZERO being the final destination. This will work if the branch is within 256 bytes. The first solution, however, is universally applicable and does not require us to insert extra lines at points relatively isolated from the flow of the program. Hence, from a general point of view, the first solution is to be preferred. Nevertheless, the second solution has the merit of being relocatable, and for stand-alone machine code programs this might be important. Which solution we use depends on our needs at the particular time. (Relocatability is discussed in more detail in Chapter 9.)

Relative addressing is justified because most branches *are* short, and the two byte instruction which relative addressing allows results in an improvement in speed and a reduction in program size. The reason for the term 'relative' should be obvious now. A branch is made x bytes relative to the present position of the program counter. Hence x is added or subtracted from the PC. By contrast, the addressing mode of JMP is absolute: the contents of the PC are replaced by the operand of JMP. Notice again how the address is stored in the form low byte, high byte in the operand. This is not so confusing if you remember that the second storage location is higher than the first in memory. We are most fortunate though that we have an assembler which sorts all this out for us, relieving our memories of unnecessary burdens.

## 4.8 USING BRANCHING IN THE ADDITION AND SUBTRACTION OF UNSIGNED NUMBERS: INC and DEC

Let us first introduce two new instructions: INC and DEC. INC MEMLOC takes the contents of MEMLOC adds one to it and stores the result in MEMLOC. If (MEMLOC) = 255, INC MEMLOC produces 0 in

MEMLOC. DEC MEMLOC takes the contents of MEMLOC subtracts one from it and stores the result in MEMLOC. If (MEMLOC) = 0, DEC MEMLOC produces 255 in MEMLOC. The mnemonic INC stands for INCrement and DEC for DECrement.

Now armed with these two new instructions let us reconsider the coding we used in listing 3.1 to add two single byte numbers together when the result may exceed 255. Here is an alternative using INC and a branch:

```
        LDA #0
        STA SUMH
        LDA NUM1
        CLC
        ADC NUM2
        STA SUML
        BCC NOCARRY
        INC SUMH
NOCARRY ________
```

The idea is if there is no carry, C = 0 and a branch occurs to line NOCARRY. Otherwise NUMH is incremented by 1. Now a comparison of the assembled form of these two programs will show that we have saved one byte (type this one in in the same way as listing 3.1 but with a two-pass assembly and check). But we do better still if we wish to encode the sum

(NUM1H; NUM1L) + (NUM2) → NUM1H; NUM1L

Using the ideas in Section 3.3 we obtain this program:

```
LDA NUM1L
CLC
ADC NUM2
STA NUM1L
LDA #0
ADC NUM1H
STA NUM1H
```

Using the new ideas here, we have instead:

```
                LDA NUM1L
                CLC
                ADC NUM2
                STA NUM1L
                BCC NOCARRY
                INC NUM1H
      NOCARRY   ————————
```

a saving now of three bytes (type in both programs and check). We also, on average, save time in execution using this second method.

Exercise 4.4

Encode (NUM1H; NUM1L) - (NUM2) → NUM1H; NUM2H into assembly language (a) using similar ideas to Section 3.6, and (b) using DEC and a branch. How many bytes are saved using (b)?

## 4.9 MONITORING PROBLEMS OF SIGN USING BRANCHING

Consider the simple BASIC statement LET DIFF = NUM1 - NUM2 where NUM1 and NUM2 are positive integers less than 255. In assembly code we have, of course:

```
                LDA NUM1
                SEC
                SBC NUM2
                STA DIFF
```

But is this really correct? What happens if (NUM1) < (NUM2)? Clearly we should get a negative result, and to accommodate this we need to treat DIFF as a signed integer. We need to take remedial action, however, if the range falls outside -128 to 127. The first thing we need to do is to assign two bytes for the difference, DIFFH, DIFFL, where (DIFFH, DIFFL) is considered a signed integer giving us a range now of -32768 to +32767, which is easily enough for our purposes.

Now there is no problem if NUM1 ⩾ NUM2; we store the result in &00; DIFFL, which is positive. What happens if NUM1 < NUM2? Let us consider an example: &83 - &C8. This gives &BB, which is to be interpreted as -69. -69 using two-byte signed integer precision is &FFBB (since &FFBB + &45 = (1)0000, and &45 is 69). It follows that the result is stored as &FF; DIFFL.

It should be clear now that the following coding will do the trick:

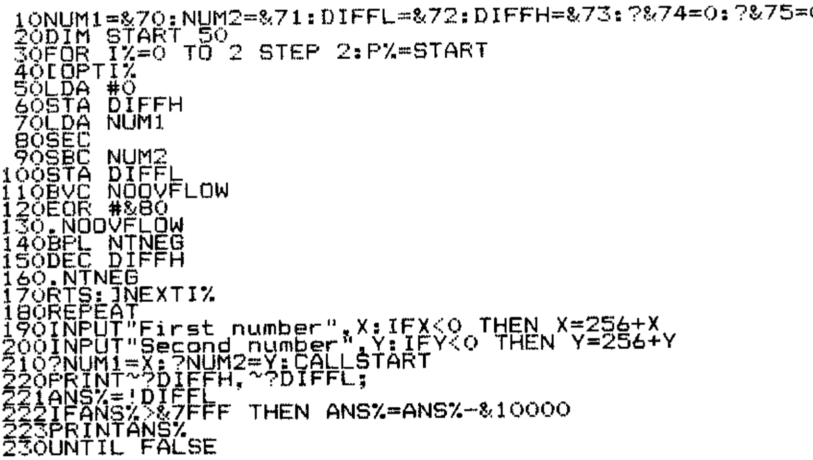
```
              LDA #0
              STA DIFFH
              LDA NUM1
              SEC
              SBC NUM2
              STA DIFFL
              BCS NTNEG
              DEC DIFFH
        NTNEG ————
```

This is very similar to the coding of the answer to Exercise 4.4. This sort of similarity is what makes the two's complement representation so useful.

Consider again the BASIC statement with which we began, but suppose now that NUM1 and NUM2 can be signed integers. If NUM1 and NUM2 have the same sign, then there is no overflow. If they were always the same sign we would, in fact, only need one byte to store the answer. This is not always the case if the signs do not agree, however. Consider 52 - (-92): here we will get overflow, since the answer will be &90 which is negative in single byte precision signed integer.

We need something similar to the solution above. The flag we need to monitor is the overflow flag. If this is not set, the sign of the answer is given by the MSB of the answer; if it is set, the sign of the answer is given by the opposite of the MSB of the answer.

LISTING 4.4



```
 10NUM1=&70:NUM2=&71:DIFFL=&72:DIFFH=&73:?&74=0:?&75=
 20DIM START 50
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60STA DIFFH
 70LDA NUM1
 80SEC
 90SBC NUM2
100STA DIFFL
110BVC NOOVFLOW
120EOR #&80
130.NOOVFLOW
140BPL NTNEG
150DEC DIFFH
160.NTNEG
170RTS:]NEXTI%
180REPEAT
190INPUT"First number",X:IFX<0 THEN X=256+X
200INPUT"Second number",Y:IFY<0 THEN Y=256+Y
210?NUM1=X:?NUM2=Y:CALLSTART
220PRINT~?DIFFH,~?DIFFL;
221ANS%=!DIFFL
222IFANS%>&7FFF THEN ANS%=ANS%-&10000
223PRINTANS%
230UNTIL FALSE
```

The coding is in listing 4.4. Notice that in line 30 we use I% = 0 and 2 giving us OPT0 and OPT2 in line 40. This suppresses the assembly translation listing completely and results in much quicker assembly. Unless we are specifically interested in seeing the translation, we should use this formulation, and we will do so from now on in this book.

EOR #&80 in line 120 reverses the MSB of the result if overflow has occurred. Notice here that we use BPL and not BCS: the carry flag has no meaning in single-byte precision signed arithmetic.

Put this program in your computer and test it with the following pairs of values for NUM1 and NUM2: -85, -18; -85, -110; -85, 30; -85, 60; 50, 20; 50, 80; 50, -30; 50, -80. Confirm that the results you get for (DIFFH) and (DIFFL) are correct.

Exercise 4.5

1. Encode the BASIC statement IF(NUM1 - NUM2) > NUM3 THEN GOTO 50 where all are unsigned integers.

2. Write the assembly code to perform

(NUM1H; NUM1L) - (NUM2H; NUM2L) → DIFF2; DIFF1; DIFF0

where the first two numbers are to be considered as unsigned integers but where (DIFF2; DIFF1; DIFF0) is a triple-precision signed integer.

3. Write the assembly code equivalent to LET NUM = NUM1 + NUM2 where NUM1 and NUM2 are integers between -128 and 127.

4. Repeat Q.2 if the first two numbers are to be considered as signed integers.

5. Write the assembly code to perform

(NUM1H; NUM1M; NUM1L) - (NUM2H; NUM2L) → NUM1H; NUM1M; NUM1L

where the first number is a negative integer but the second is not signed. Include a check for overflow.

6. Repeat Q.3 if NUM1 and NUM2 are integers between -32768 and 32767 whose sum may lie outside this range without an error being signalled.

7. Write the coding to branch if NUM1 ⩾ NUM2 where both are signed integers.

8. Repeat Q.1 if NUM1 and NUM2 can be signed integers, but where NUM3 remains unsigned.

9. Repeat Q.1 if all the numbers are signed integers.

10. Repeat Q.7 if the numbers are each two-byte signed integers.

# Chapter 5 Loop structure in assembly language

## 5.1 LOOP STRUCTURES

BBC BASIC has two loop structures: the FOR... NEXT loop and the REPEAT... UNTIL loop. These structures are essential components of most realistic programs in BASIC, and the same is true of assembly language. However, in assembly language the structures do not come ready made as in BASIC, so we need to develop methods of building them up from the component instructions available. In doing this, we will also create a third important structure, regrettably missing in BBC BASIC: this structure we will term the REPEAT WHILE loop. The difference between REPEAT... UNTIL and REPEAT WHILE is that the former will always be executed at least once, since the exit is at the end; the latter need not be executed at all, since the exit is at the beginning.

The most important instructions needed to create loop structures in 6502 assembly language are connected with the X and Y registers, and to these we now turn.

## 5.2 INDEX REGISTERS: SOME NEW INSTRUCTIONS

There are two further registers in the 6502 microprocessor which can act as temporary storage locations of data: the X register and the Y register. Their relationship to the rest of the architecture can be seen in the diagram in Appendix 2.

Data can be loaded into them from memory: the mnemonics are LDX and LDY. Data can be stored from them into memory: the mnemonics are STX and STY. There are also instructions identical in operation to CMP: CPX M compares the contents of X with (M) and sets the appropriate flags; similarly CPY M.

A very important feature of the registers is that they can be incremented and decremented directly: the mnemonics are INX, INY, DEX, DEY. These are one-byte instructions utilising implied addressing and so they save space over a corresponding INC M or DEC M instruction (which is at least two bytes long). More importantly, they operate at least 2½ times as fast as the corresponding INC M and DEC M operations. This is very telling in loops, as we shall see.

No arithmetic or logical operations can take place on the X and Y registers: there is no equivalent to ADC, SBC, ORA, EOR or AND.

Hence to add a number to X or Y we must first transfer X or Y to the accumulator, perform the addition or subtraction, and then transfer back. For this reason, there are special instructions allowing us to make these transfers: the mnemonics are TXA (transfer contents of X to A), TYA, TAX (transfer contents of A to X), TAY. All are one byte instructions.

## 5.3 THE ASSEMBLY LANGUAGE EQUIVALENT OF A FOR... NEXT LOOP

(a) FOR X = 1 TO 40 .......... NEXT

How can we implement this structure in assembly language? Here is the coding:

```
        LDX #1
LOOP    .
        .
        .
        .
        .
        INX
        CPX #41
        BNE LOOP
```

We begin by setting the X register to 1. Then the first cycle of the loop is performed; that is, any instructions down to INX will be obeyed. On reaching INX, X is incremented and compared to 41; if the result is not zero the next cycle of the loop will be performed. Altogether there will be 40 cycles through the loop, and you should convince yourself of this.

(b) FOR X = 40 to 1 STEP -1 .......... NEXT

This is more complicated in BASIC but easier in assembly language:

```
        LDX #40
LOOP    .
        .
        .
        .
        .
        DEX
        BNE LOOP
```

There is a saving of two bytes and a corresponding saving of time. This is because the decrement operation affects the Z flag (so too does the increment operation - as X or Y passes from 255 to 0 again - and both operations also affect the N flag). Hence there is no need for a CPX #0 after DEX.

Saving time is much more important in loops than in parts of a program which will only be performed once. If possible, then, we should try to organise our loops to count backwards as here, rather than forwards as in (a). And we should always use the X (or Y) register as the loop counter, since INX and DEX operate very much more quickly than INC and DEC.

(c) FOR X = 0 TO NUM .......... NEXT

```
             LDX #0
       LOOP   .
              .
              .
              .
              .
             INX
             BEQ OUT
             CPX NUM
             BCC LOOP
             BEQ LOOP
        OUT
```

Here X is compared with NUM, and the loop is performed if X is less than NUM (BCC LOOP). This will give (NUM) cycles of the loop, and one more is gained when X equals (NUM) (BEQ LOOP). Notice that we do not use BMI LOOP, tempting though it may be: if (NUM) is 130 or more, the loop would only be performed once! Finally, the purpose of BEQ OUT: without it, if (NUM) = 255, the loop would cycle indefinitely.

(d) FOR X = NUM TO 0 STEP -1 ......... NEXT

As before, this is more efficient than the forward loop, but for a different reason.

```
             LDX NUM
       LOOP   .
              .
              .
              .
              .
             DEX
             CPX #&FF
             BNE LOOP
```

This time we still need a CPX, but we cannot use CPX #0: BCS LOOP since this will give us an infinite loop! Again, trying to do without the CPX and writing just BNE LOOP will miss out the last cycle of the loop (when X = 0). Hence we need to compare X to one less than zero i.e. &FF. But we save two bytes on needing only one test instead of two.

You might be wondering why we *cannot* write

```
          LDX NUM
LOOP      .
          .
          .
          .
          .
          DEX
          BPL LOOP
```

since DEX affects the N flag, and so save ourselves two more bytes. The problem is similar to that discussed in (c): if NUM is greater than 128, the loop will only be performed once.

Comparing the programs in (b) and (d) we see that it is more efficient to count down to 1 than to zero. Hence, if we can, we should try to organise our loop to count backwards to 1 rather than to zero.

(e) FOR X = NUM1 TO NUM2 .......... NEXT

If we assume NUM2 ≥ NUM1 we can write:

```
          LDX NUM1
LOOP      .
          .
          .
          .
          .
          INX
          BEQ OUT
          CPX NUM2
          BCC LOOP
          BEQ LOOP
OUT
```

What happens here if NUM1 < NUM2? Clearly, the loop is performed once. Most BASICs, including BBC BASIC, are like this.

(f) FOR X = NUM1 TO NUM2 STEP NUM3 .......... NEXT

```
             LDX NUM1
       LOOP   .
              .
              .
              .
              .
             TXA
             CLC
             ADC NUM3
             TAX
             BCS OUT
             CMP NUM2
             BCC LOOP
             BEQ LOOP
        OUT
```

To add on (NUM3) we need to transfer X to the accumulator (TXA), perform the addition and transfer back (TAX). We can now use CPX NUM2 or CMP NUM2, it does not matter. The statement BCS OUT guards against a case like FOR X = 6 TO 251 STEP 5, where X returns to zero before it can exceed 251.

Note that all of (a) to (f) will work equally well if Y replaces X.

Exercise 5.1

1. Will the programs in (e) and (f) need any amendment if we interpret the numbers as signed integers (i.e. NUM1, NUM2 and NUM3 lie between -128 and 127)?

2. Assuming we are dealing with unsigned integers write in assembly language:

(a) FOR X = NUM2 TO NUM1 STEP -1 .......... NEXT

(b) FOR X = NUM2 TO NUM1 STEP -(NUM3) .......... NEXT

Take care if NUM1 = 0 in (a), and guard against something similar to 5.3f in (b).

3. Amend the program in (f) in line with the suggestions in the answer to Q.1 if the numbers are signed integer.

## 5.4 FOR... NEXT LOOPS OF MORE THAN 256 CYCLES

### (a) FOR N = 1 to 1000 .......... NEXT

To implement this in assembly language we need to use both the X and the Y counters. In general we choose for X the largest whole number less than 257 which will divide into the cycle count. Here we take 250 for X and 4 for Y as follows:

```
      LDY #0
LOOP1 LDX #1
LOOP2  .
       .
       .
       .
       .
      INX
      CPX #251
      BNE LOOP2
      INY
      CPY #4
      BNE LOOP1
```

The inner loop, LOOP2 to BNE LOOP2 is performed 250 times for each of the four values of Y (0, 1, 2 and 3).

To obtain the value of N, contained in two bytes of course, we would have to evaluate 250 * Y + X. We will show how to do this in Chapter 8.

### (b) FOR N = 1000 TO 1 STEP -1 .......... NEXT

Once again, the backwards loop is quicker.

```
      LDY #4
LOOP1 LDX #250
LOOP2  .
       .
       .
       .
       .
      DEX
      BNE LOOP2
      DEY
      BNE LOOP1
```

This time N is 250 * (Y - 1) + X. It follows that if the value of N is required on each cycle of the loop, then there is less saving in moving backwards through the loop. Having to subtract 1 from Y cancels out some of the time otherwise saved in the implementation of the loop. We can in this case alter the program so that N is 250 * Y + X. Alter the first line to LDY #3 and the last to BPL LOOP1. This is not a general solution but will work if the initial content of Y is less than 129.

(c) To generalise this for N = 0 TO (NUMH; NUML) we will need to use a different method. In any specific case we can examine the number of cycles and write the program as in (a) (or (b)). But, if we want to accommodate any number of cycles up to 65536, it is tricky to incorporate these factor type calculations in the program. An easier method is to separate the loop into two parts: N = 0 TO {(NUMH) - 1};FF and N = (NUMH);00 TO (NUMH; NUML).

The first loop is achieved by cycling on X 256 times for each value of Y = 0, 1, 2... (NUMH) - 1; the second by cycling on X for 0 to (NUML).

Here is the program:

```
         LDX #0
         LDY #0
LOOP1     .
          .
          .
          .
          .
         INX
         BNE LOOP1
         INY
         CPY NUMH
         BNE LOOP1
LOOP2     .
          .
          .
          .
          .
         INX
         BEQ OUT
         CPX NUML
         BCC LOOP2
         BEQ LOOP2
   OUT
```

X in LOOP1 goes from zero back to zero, giving 256 cycles; Y goes from 0 to (NUMH) - 1, one for each complete loop of X. Then in LOOP2, X goes from zero to NUML. At any stage N = (Y; X).

It may seem rather wasteful to have two separate loops like this, but as we shall see in Chapter 9, if a lot of code is required within the loop then a subroutine can be used. In this way, the memory overhead is not so great (though the subroutine solution will add a time cost).

This method is quite general: if we want to perform N = A TO B we use instead N = 0 TO (B - A), and compute N as (Y; X) + A.

Exercise 5.2

1. What happens to the program in (c) if NUMH is zero? Write some code to correct this.

2. Write the program in (c) using a 'backwards' loop, N = (NUMH; NUML) TO 0 assuming (i) that N will not need to be computed (so that no strict order on N is necessary) and (ii) that N needs to be computed on every loop in the correct coding order. Are both programs an improvement over the forward loop?

## 5.5 THE EQUIVALENTS OF A REPEAT... UNTIL AND A REPEAT WHILE... ENDWHILE LOOP

(a) REPEAT .......... UNTIL SUM > 200

There are two ways of coding this:

```
(i) REPUNTIL     .             (ii) REPUNTIL     .
                 .                               .
                 .                               .
                 .                               .
                 .                               .

          LDA SUM                          LDA #200
          CMP #200                         CMP SUM
          BCC REPUNTIL                     BCS REPUNTIL
          BEQ REPUNTIL
```

Clearly the second is more efficient, and illustrates a general principle: if possible, arrange the program so that a branch will occur when the value in the accumulator is greater than or equal to the value compared (or alternatively when the value in the accumulator is less than the value compared). Try to avoid branches on 'greater than's' or on 'less than or equal to's'. The same is true if we want to compare a value in the X or Y register with a value in memory or in immediate mode. Nevertheless, do not attempt to distort the flow of logic to achieve this, since you will end up with a less efficiently organised and hence slower program. Only if the choice of comparison can be made without any real alteration to the natural flow in the algorithm, should you contemplate it.

(b) REPEAT WHILE SUM < 200 .......... ENDWHILE

As mentioned this structure does not exist in BBC BASIC, but it should be clear what it does. ENDWHILE marks the end of the group of instructions to be repeated while sum remains less than 200.

In assembly code:

```
REPWHILE LDA #200
         CMP SUM
         BCC ENDWHILE
               .
               .
               .
               .
         JMP REPWHILE
ENDWHILE
```

In practice it may be possible to use a branch rather than a jump here (see 5.7c, for example). Notice the meaningful label names chosen. It is possibly good practice to stick to these names in all your programs, adding numbers if necessary to differentiate. So for example, you may have the labels REPUNTIL1, REPUNTIL2, REPWHILE1, ENDWHILE1, REPWHILE2, ENDWHILE 2, LOOP1, LOOP2, LOOP3 in one program. Remember the BBC assembler allows you to have label names of any length.

## 5.6 ARITHMETIC AND LOGICAL OPERATIONS CONCERNING THE X AND Y REGISTERS

This section looks at some special techniques concerned with performing arithmetic on the X and Y registers. In all cases, where X is referred to, the same will hold for Y.

(a) (M) + X → M

Recall that this means that the contents of the memory location whose address is M is added to the X register (X needs no brackets round it for its address is implicit and so X can unambiguously refer to the contents of the register). The result is to be put in the location with address M.

```
TXA
CLC
ADC M
STA M
```

We use TXA to economically transfer X to the accumulator where the arithmetic is performed.

(b) (M) - X → M

We cannot use the last method for this, since it would produce X - (M). It may seem that we need a temporary location to do this, thus:

```
LDA M
LDX TEMP
SEC
SBC TEMP
STA M
```

but this is unnecessary, in fact.

Instead we can write:

```
LDA M
STX M
SEC
SBC M
STA M
```

Once we have put the original value of (M) into the accumulator the location M can be used as a temporary store for X.

(c) A + X → A

Here it would seem that we have no alternative to a temporary location:

```
STX TEMP
CLC
ADC TEMP
```

However, if the program is written in RAM, as all the BBC programs you write will be, we can use the area of program itself as a temporary location, so economising on locations needed. The coding is:

```
       STX MEMLOC +1
       CLC
MEMLOC ADC #0
```

MEMLOC +1 is the address of the location where the value 0 is currently stored (this value of zero is a dummy value and could be any number between 0 and 255). Hence the value of X is stored as the operand of ADC 'immediate' as required. (Note that MEMLOC +2, MEMLOC +10 or even MEMLOC +I are possible too, if we need them: indeed any expression can be used as an operand and will be evaluated by the same arithmetic routines as the BASIC interpreter uses.) The main problem with this method is that it results in non relocatable code (see Chapter 9). So if you wish a particular stand-alone machine code program to be relocatable you will have to use a temporary location. A second problem is that we lose the benefits of zero page storage, and this gives a small time cost.

(d) X - A → A

Using the idea in the last section, we can do without a temporary location external to the program:

```
       STA MEMLOC +1
       TXA
       SEC
MEMLOC SBC #0
```

The value in the accumulator becomes the operand of SBC#, since it is stored in location MEMLOC +1, the address of the value zero. Then X is transferred to the accumulator and the old value of A is subtracted from it.

(e) A : X?

This notation may be new to you. It means that A is to be compared to X, and a decision is to be made depending upon this comparison. In the context of assembly language, X is taken from A and the appropriate flags are set; the contents of A and X remain unchanged.

We can again use the ideas in (c):

```
       STX MEMLOC +1
MEMLOC CMP #0
```

This compares the accumulator to the X register as required.

(f) Exchange X and Y without affecting A
or any other external memory locations

Unfortunately there is no 6502 instruction to do this important task. Nevertheless, we can again use the ideas in (c) to produce a compact solution:

```
        STX MEMLOC2 +1
        STY MEMLOC1 +1
MEMLOC1 LDX #0
MEMLOC2 LDY #0
```

Exercise 5.3

1. Write assembly code to perform the following operations as economically as possible:

(a) X - (M) → M

(b) A - X → A

(c) X - (M) → X

(d) A - X → X

(e) X + Y → A

(f) X - Y → A

(g) X : A?

(h) X : Y?

(i) exchange X and A

2. Write code to perform X + (M) → M which at the end of the operation leaves the contents of A and Y unchanged.

3. Rewrite (c) and (d) of Q.1 leaving the contents of A and Y unchanged at the end.

4. Write code to perform X + Y → X, leaving A and Y unchanged at the end.

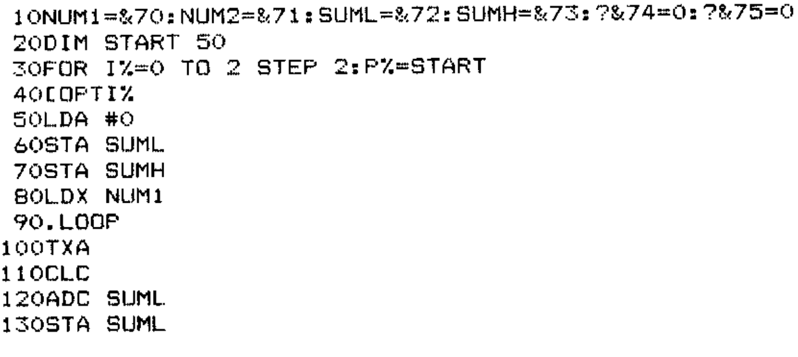
## 5.7 SOME EXAMPLE PROGRAMS USING LOOP STRUCTURE

We consider now some examples utilising the foregoing ideas.

(a) To find the sum of the numbers from (NUM1) to (NUM2) inclusive ($0 \leqslant$ (NUM1) $<$ (NUM2) $< 256$).

We use the structure discussed in 5.3e. The result of the addition is stored in SUMH; SUML (two bytes will be adequate since the sum must be less than $256^2$). The program is in listing 5.1.

LISTING 5.1



```
 10NUM1=&70:NUM2=&71:SUML=&72:SUMH=&73:?&74=0:?&75=0
 20DIM START 50
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60STA SUML
 70STA SUMH
 80LDX NUM1
 90.LOOP
100TXA
110CLC
120ADC SUML
130STA SUML
140BCC NOCARRY
150INC SUMH
160.NOCARRY
170INX
180BEQ OUT
190CPX NUM2
200BCC LOOP
210BEQ LOOP
220.OUT
230RTS:]NEXTI%
240CLS:REPEAT
250INPUT"Sum from",?NUM1
260INPUT"To",?NUM2
270CALLSTART
280PRINT!SUML
290UNTIL FALSE
```

(b) To find how many natural numbers from 1 must be added together for the sum to exceed (STOTAL) (0 ⩽ (STOTAL) ⩽ 255).

We use the structure in 5.5a and the program is given in listing 5.2. The number of numbers required will be contained in X, the interim result in SUM. Notice the importance of line 140: without this line, if (STOTAL) = 255, for example, there would never be an exit from the loop.

LISTING 5.2

```
 10SUM=&70:STOTAL=&71
 20DIM START 50
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60STA SUM
 70TAX
 80.REPUNTIL
 90INX
100TXA
110CLC
120ADC SUM
130STA SUM
140BCS OUT
150CMP STOTAL
160BCC REPUNTIL
170BEQ REPUNTIL
180.OUT
190RTS:]NEXTI%
200CLS:REPEAT
210INPUT"Maximum total",?STOTAL
220PRINT((USRSTART AND &0FFFFFFF)MOD &10000) DIV &100
230UNTIL FALSE
```

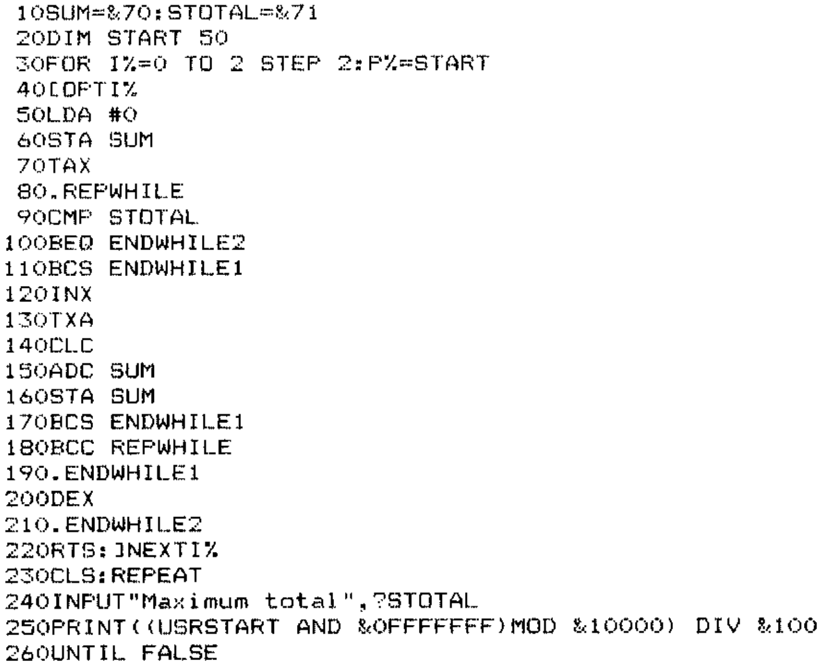
Line 220 isolates the second byte from the left of USRSTART, which is the contents of the X register. The purpose of AND is to make sure the MSBit is zero; if it is 1, the technique will not work. An alternative way of isolating bytes in the USR function is given in 6.5.

Unlike CALL, USR always returns a value and must either be printed out or put equal to another variable. Its principal function is to return one or more of the registers' values. The value returned consists of four bytes in fact, PYXA, most significant to least significant, where P is the processor status register. We will have more to say about USR in Chapter 6.

(c) To find the greatest number of natural numbers from 1 which can be added together with their sum not exceeding (STOTAL) (0 ⩽ (STOTAL) ⩽ 255).

In this case, we need the structure in 5.5b, since if (STOTAL) = 0 we do not want to perform the loop at all. Listing 5.3 contains the program.

LISTING 5.3



```
 10SUM=&70:STOTAL=&71
 20DIM START 50
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60STA SUM
 70TAX
 80.REPWHILE
 90CMP STOTAL
100BEQ ENDWHILE2
110BCS ENDWHILE1
120INX
130TXA
140CLC
150ADC SUM
160STA SUM
170BCS ENDWHILE1
180BCC REPWHILE
190.ENDWHILE1
200DEX
210.ENDWHILE2
220RTS:]NEXTI%
230CLS:REPEAT
240INPUT"Maximum total",?STOTAL
250PRINT((USRSTART AND &0FFFFFFF)MOD &10000) DIV &100
260UNTIL FALSE
```

Notice that there are two exits to the loop: ENDWHILE1 and ENDWHILE2. We exit to the former if (SUM) exceeds (STOTAL): in this case we must reduce X by 1, since we require the sum of the X's not to exceed (STOTAL). We exit to the latter if (SUM) equals (STOTAL): in this case, we have not exceeded (STOTAL) and so X does not need to be decremented.

## Exercise 5.4

1. Write a program to obtain the sum of 1 + 2 + 4 + 7 + 11 + ... to NUM ($\leqslant$ 256) terms. Put the result in SUM3; SUM2; SUM1. Store each term of the series in TERMH; TERML.

2. Write a program to find how many terms (< 256) of the series in Q.1 must be added together for the sum to exceed STOTAL3; STOTAL2; STOTAL1.

3. Write a program to find the greatest number of the terms (< 256) of the series in Q.1 which can be added together with their sum not exceeding STOTAL3; STOTAL2; STOTAL1.

# Chapter 6 Indexed addressing

## 6.1 MOVING A SECTION OF MEMORY

Consider the following fragment of a BASIC program:

```
100 FOR X = (NUMBER - 1) TO 0 STEP - 1
110 ? (NEWLOC + X) = ? (OLDLOC + X)
120 NEXT
```

Its purpose is to transfer a number of bytes from one section of memory to another. The starting location of the bytes to be moved is OLDLOC and the first location of the point to which they are to be moved is NEWLOC. Altogether, NUMBER bytes are transferred.

It would be very useful if we could write a program in assembly code to do this with the same structure as the BASIC program. Fortunately, an addressing mode exists which allows us to do this: it is called *indexed addressing*. Here is the assembly program:

```
     LDX NUMBER
     DEX
LOOP LDA OLDLOC, X
     STA NEWLOC, X
     DEX
     CPX #&FF
     BNE LOOP
```

The new addressing mode is at lines 3 and 4: LDA OLDLOC, X and STA NEWLOC, X. The first loads the accumulator with the contents of the memory location whose address is OLDLOC + X; it is therefore equivalent to the right hand side of line 110 in the BASIC program. The second stores the accumulator in NEWLOC + X, equivalent to the left hand side of line 110.

The purpose of indexed addressing is to allow access to one of a series of up to 256 continuous bytes, the base address of which is known. Figure 6.1 illustrates this, where the base address is given the label BASMEM.

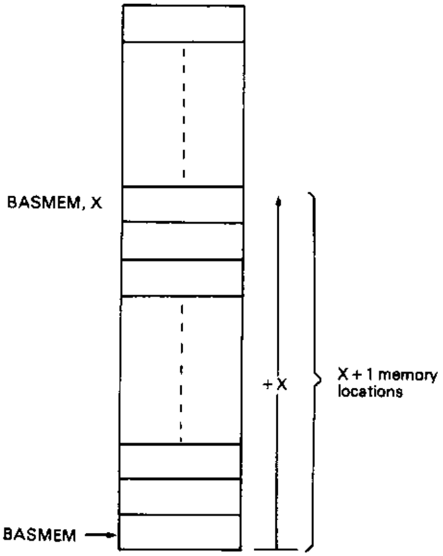


*Figure 6.1 Indexed addressing*

## 6.2 IMPROVING THE PROGRAM

The program in 6.1 can be improved upon. As it stands we need to compare X to &FF at each stage of the loop, which takes time. By rewriting the program, we can eliminate this step:

```
     LDX NUMBER
LOOP LDA OLDLOC - 1, X
     STA NEWLOC - 1, X
     DEX
     BNE LOOP
```

When this is assembled, 1 will be subtracted from the values of OLDLOC and NEWLOC. Hence on the first cycle of the loop, we deal with locations OLDLOC - 1 + NUMBER and NEWLOC - 1 + NUMBER; and on the last cycle, when X is 1, we deal with OLDLOC and NEWLOC themselves. In this way we save three bytes and, more importantly, significant processing time. Incidentally, don't be tempted to put brackets round OLDLOC - 1 and NEWLOC - 1. As we saw in an earlier chapter, brackets round addresses are interpreted as referring to the *contents* of those addresses by the assembler. This will turn out to be a useful convention in the next chapter, when we see how to access more than 256 continuous bytes.

Our program will work in all cases, except where an early part of the old locations overlap a later part of the new locations i.e. where NEWLOC < OLDLOC but NEWLOC - 1 + NUMBER ≥ OLDLOC.

Figure 6.2 illustrates a typical case. It can be seen that, for example, locations C and D will lose their contents (to those of A and B) before they have been transferred to their new homes (at E and F). This problem can be solved by moving forwards through the loop instead of backwards. Here is the program:

```
     LDX #0
LOOP LDA OLDLOC, X
     STA NEWLOC, X
     INX
     CPX NUMBER
     BNE LOOP
```

This method inevitably requires the CPX instruction, and so executes in a longer time than the first program.

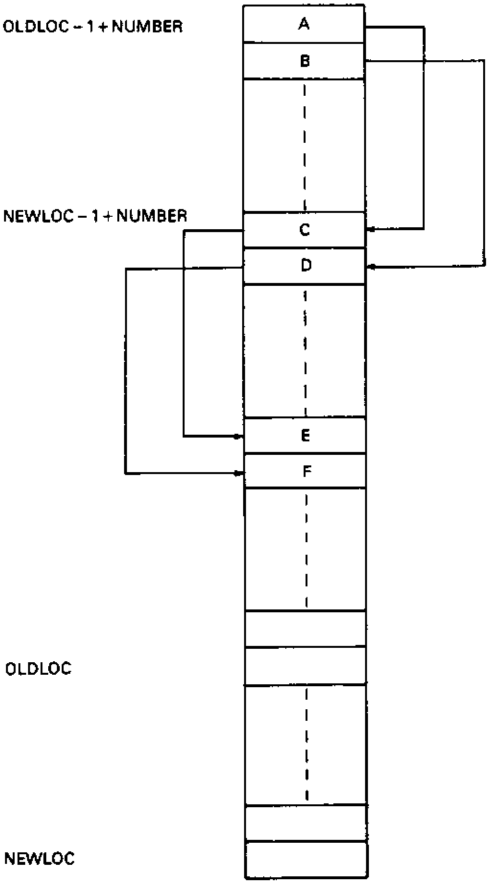


*Figure 6.2: Where moving backwards through a loop will not work.*

Exercise 6.1

Under what circumstances will the 'moving forwards' method not work?

## 6.3 THE RANGE OF INSTRUCTIONS FOR WHICH INDEXED ADDRESSING IS AVAILABLE

All that we have said above applies equally to the Y register, but it turns out that the X-register indexed addressing mode has a much greater range of application than the Y-register indexed mode.

Some instructions allow both indexed-X and indexed-Y modes: ADC, AND, CMP, EOR, LDA, ERA, SBC and STA. But there are some which allow only indexed-X modes: INC, DEC, LDY and the new instructions to be covered in Chapter 8. The byte- and time-saving aspects of zero page storage are only available with the X-register: with indexed-Y, zero page must be treated as absolute; so, for example, LDA &70, Y will be translated as B9 70 00 by the assembler, whereas LDA &70, X will become B5 70. Moreover, zero-paged indexed-X operates circularly: e.g. LDA &73,X when X is &FD will access location &70 (i.e. since &73+&FD>&FF,&FD is treated as -3). This means you can move forwards through a loop ending at X=0: e.g. LDX#&FD: LOOP:LDA &73,X:INX:BNE LOOP, will load from locations &70 to &72 inclusive. There are a few cases where zero page-Y is allowed: LDX and STX both allow this mode (and obviously do not allow indexed-X modes), and in the case of STX this is the only indexed mode available, absolute-Y not being permitted either (a similar constraint applies to STY where zero page-X is the only indexed mode). There are just three instructions which do not allow any indexed addressing: CPX, CPY and BIT (to be covered in Chapter 9). All of this information is contained in tabular form in Appendix 1.

## 6.4 ARRAYS

You will know that in BBC BASIC you can declare an integer array by writing simply, say, DIM ARRAY% (99). This produces 100 elements labelled ARRAY% (0) to ARRAY% (99), each one initially having the value zero.

Can we do a similar thing in assembly language? Here is the program:

```
     LDX #100
     LDA #0
LOOP STA ARRAY - 1, X
     DEX
     BNE LOOP
```

ARRAY being the address of the first location of the memory.

Now, we have not, in fact, done exactly the same thing as in BASIC, since ARRAY% (99) allocates *four* bytes to each element of the array. These four bytes give a range of $-2^{31}$ to $2^{31} - 1$, using signed arithmetic. Our program is in fact equivalent to the BASIC statement DIM ARRAY 99, which allocates 100 bytes, the first being at address ARRAY (which the interpreter will allocate as the first free location above TOP). To manipulate these bytes in BASIC, we would have to use statements like ?(ARRAY + X) = TEMP. This is a little clumsy in BASIC, but for our purposes it has the advantage of being very close to assembly language, so let us examine a very simple (and inefficient) sorting program, to sort the 100 bytes into numerical order, smallest first.

First the program in BASIC: see Listing 6.1.

LISTING 6.1

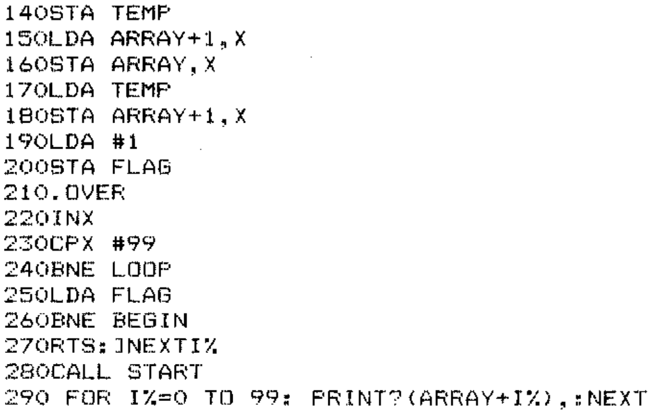
```
 10DIM ARRAY 99
 20 FOR I%=0 TO 98 STEP4:!(ARRAY+I%)=RND:NEXT
 30FLAG=0
 40FOR I%= 0 TO 98
 50 IF ?(ARRAY+I%)<=?(ARRAY+I%+1)THEN100
 60TEMP=?(ARRAY+I%)
 70?(ARRAY+I%)=?(ARRAY+I%+1)
 80?(ARRAY+I%+1)=TEMP
 90FLAG=1
100NEXTI%
110IF FLAG=1 THEN30
120 FOR I%=0 TO 99: PRINT?(ARRAY+I%):NEXTI%
```

The idea is to scan through the 100 bytes and swop any contiguous pair of bytes where the lower byte is greater than the upper byte. Lines 60 to 80 perform the swop. The process is repeated until no swops are necessary, when the sorting is complete: when this happens, FLAG will be zero on leaving the for/next loop at 100. Lines 10 and 20 create an array of 100 random bytes, and line 120 prints out the results.

Thus us now construct an assembly language program to do this: Listing 6.2.

LISTING 6.2

```
 10DIM ARRAY 99 :DIM START 50
 20 FOR I%=0 TO 98 STEP4:!(ARRAY+I%)=RND:NEXT
 30FLAG=&70:TEMP=&71
 40FORI%=0 TO 2 STEP 2:P%=START
 50[OPTI%
 60.BEGIN
 70LDX #0
 80STX FLAG
 90.LOOP
100LDA ARRAY,X
110CMP ARRAY+1,X
120BCC OVER
130BEQ OVER
```



```
140STA TEMP
150LDA ARRAY+1,X
160STA ARRAY,X
170LDA TEMP
180STA ARRAY+1,X
190LDA #1
200STA FLAG
210.OVER
220INX
230CPX #99
240BNE LOOP
250LDA FLAG
260BNE BEGIN
270RTS:]NEXTI%
280CALL START
290 FOR I%=0 TO 99: PRINT?(ARRAY+I%),:NEXT
```

The details are:

| | | | | | |
|---|---|---|---|---|---|
| 80 | As | line | 30, | Listing | 6.1 |
| 100-130 | " | " | 50 | " | " |
| 140 | " | " | 60 | " | " |
| 150-160 | " | " | 70 | " | " |
| 170-180 | " | " | 80 | " | " |
| 190-200 | " | " | 90 | " | " |
| 220-240 | " | " | 100 | " | " |
| 250-260 | " | " | 110 | " | " |

It is instructive to run both programs and compare the time taken! (Actually, the dice are weighted even more against BASIC since ARRAY is not an integer variable; changing this to ARRAY% speeds up the program a little.)

Now, whilst modelling our assembly program on a BASIC program makes it a little easier to write, it does not generally produce the most economical and efficient coding. In this case, a saving will be produced if we count backwards from 99 to zero, and this is left as an exercise in Exercise 6.2. (A saving also occurs if we use the stack for temporary storage as we shall see in Chapter 9.)

An analysis of the algorithm will point to an immediate improvement. On the first scan of the array, the largest value will end up in ARRAY + 99. We can therefore now ignore this and scan from ARRAY to ARRAY + 98: the next largest value will now be in ARRAY + 98. Now scan from ARRAY to ARRAY + 97 etc. The BASIC program for this is in Listing 6.3.

LISTING 6.3

```
   10DIM ARRAY 99
   20 FOR I%=0 TO 98 STEP4:!(ARRAY+I%)=RND:NEXT
   30FOR J%=1 TO 99
   40FOR I%=0 TO 99-J%
   50 IF ?(ARRAY+I%)<=?(ARRAY+I%+1)THEN90
   60TEMP=?(ARRAY+I%)
   70?(ARRAY+I%)=?(ARRAY+I%+1)
   80?(ARRAY+I%+1)=TEMP
   90NEXTI%
  100NEXTJ%
  110 FOR I%=0 TO 99: PRINT?(ARRAY+I%):NEXTI%
```

This method of sorting is called a bubble sort.

The assembly program is in Listing 6.4.

LISTING 6.4

```
   10DIM ARRAY 99 :DIM START 50
   20 FOR I%=0 TO 98 STEP4:!(ARRAY+I%)=RND:NEXT
   30TEMP=&70
   40FORI%=0 TO 2 STEP 2:P%=START
   50[OPTI%
   60LDY #99
   70.BEGIN
   80LDX #0
   90STY MEMLOC+1
  100.LOOP
  110LDA ARRAY,X
  120CMP ARRAY+1,X
  130BCC OVER
  140BEQ OVER
  150STA TEMP
  160LDA ARRAY+1,X
  170STA ARRAY,X
  180LDA TEMP
  190STA ARRAY+1,X
  200.OVER
  210INX
  220.MEMLOC
  230CPX #0  (Dummy operand)
  240BNE LOOP
  250DEY
  260BNE BEGIN
  270RTS:]NEXTI%
  280CALL START
  290 FOR I%=0 TO 99: PRINT?(ARRAY+I%),:NEXT
```

We have used the Y register here to store the value (99 - J%) + 1, which starts at 99 for the first for/next loop and ends at one for the last. At each stage we compare X to value of Y, using the device discussed in the last chapter to do this.

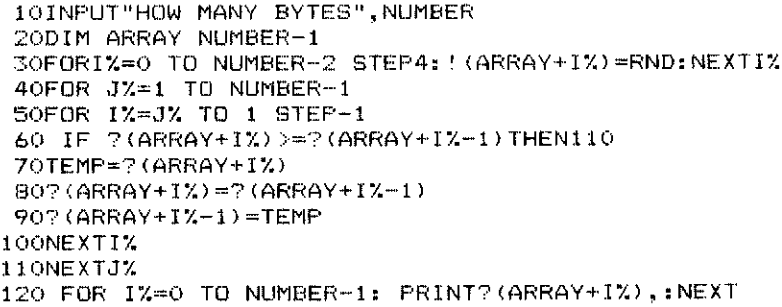
The bubble sort is not the most efficient way of sorting (for example, a method called Shell-Metzner is better), but machine code is so quick that this is not really important here.

## Exercise 6.2

1. Improve the first assembly program by counting backwards, at the same time generalising it to allow the sorting of NUMBER bytes (NUMBER ≤ 256), making any other changes which improve economy. Explain why it is not necessary to treat NUMBER as an address in this case.

2. Repeat Q.1 for the second assembly program. Notice that this time, the *smallest* value drops through to the *first* element of the scan e.g. after the first cycle, ARRAY will contain the smallest element. Is this design more or less efficient than the original design?

3. The following BASIC program is another type of sort (an insertion sort). Encode it into assembly language in the most economical way.



```
 10INPUT"HOW MANY BYTES",NUMBER
 20DIM ARRAY NUMBER-1
 30FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
 40FOR J%=1 TO NUMBER-1
 50FOR I%=J% TO 1 STEP-1
 60 IF ?(ARRAY+I%)>=?(ARRAY+I%-1)THEN110
 70TEMP=?(ARRAY+I%)
 80?(ARRAY+I%)=?(ARRAY+I%-1)
 90?(ARRAY+I%-1)=TEMP
100NEXTI%
110NEXTJ%
120 FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

## 6.5 A FUNDAMENTAL DATA STRUCTURE: THE QUEUE

The queue is a familiar idea in everyday life: you go into a shop, but the shopkeeper is busy, so you form a queue. The person at the head of the queue, the person who has been there longest, gets served first. The principle is first in, first served.

Sometimes the microprocessor is very busy on important tasks, but data keeps arriving from some source. The microprocessor wants to store this data away until it has time to process it, and it wants to deal with it on a first come, first served principle. Thus it wants to form a queue of this data, and when it has the time it can deal with it. This is termed a first in, first out (FIFO) data structure, for obvious reasons.

Now it would seem very simple to deal with this. Just set 256 bytes of memory aside, and use an index register to point to the first free position in the queue as data arrives. When time allows, the microprocessor starts processing the data from the first item in memory upwards. Thus the situation is as in Figure 6.3, where BASMEM is the lowest memory position and X points to the first free location in the queue. Then, STA BASMEM, X will put the next item of data in the memory, and INX will update the pointer to the free position. And when time permits, the microprocessor can access the data by performing the cycle LDA BASMEM, Y : INY, for Y taking values from 0 to X - 1.

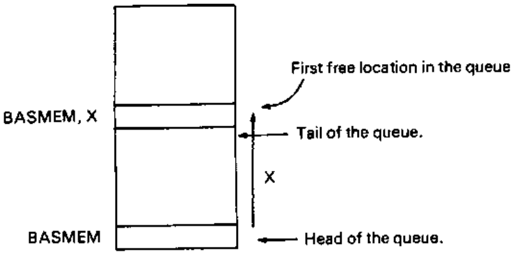


*Figure 6.3: A very simple queue implementation.*

A problem arises, however, if the microprocessor does not have enough time to access the whole queue. It may get half way through and have to go back to its more important tasks. But now more data may again arrive, and there is a danger that space will run out, even though the lower part of the memory allocation is actually free. One way round this is to move the remaining queue 'up', so that the head of the queue is again at BASMEM. Figure 6.4 illustrates this.

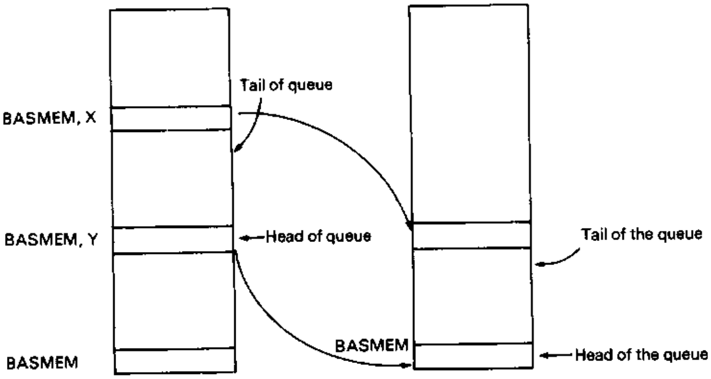


*Figure 6.4: One solution to the memory wastage problem.*

This idea is not very sensible, however. The microprocessor is not accessing the whole queue because it does not have time: it can hardly have time, then, to move up the entire queue! What is required is a way of always using all 256 locations without moving around memory; and this can be done by treating the memory as a circular and not linear series of 256 locations.

We need to use the X register to point to the first free location as before, and we use the Y register to point to the head of the queue. Thus STA BASMEM, X : INX will deposit a new item of data and adjust the tail pointer; and LDA BASMEM, Y : INY will withdraw the item at the head of the queue and adjust the head of queue pointer.

We begin with X = Y = 0. Data comes in and is put on the queue. The situation is now as in Figure 6.3: Y is still 0, but X points to the next free location. The microprocessor gets some time and starts to access the queue, but before it finishes it has to go back to more important tasks. The situation is now as in the left hand side of Figure 6.4: Y points to the new head of the queue; and X points to the tail (which may be longer still, since data may have been entered on the queue while the microprocessor was accessing the head).

Suppose now that a large amount of data comes in so that X reaches 255. At this stage X resets to zero. More data still comes in, and the situation when the microprocessor is again ready to access the queue is as in Figure 6.5. We see that Y is greater than X but this does not matter. The same instruction - LDA BASMEM, Y : INY - will withdraw the data one at a time; and this will continue to be so as Y increases through 255 to zero.

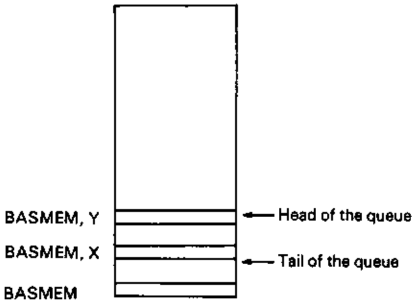


*Figure 6.5: A better solution to the memory wastage problem.*

If at any stage X = Y then either the queue is empty or it is full. Which it is will depend on whether the last transaction was a withdrawal or a deposit. The microprocessor will need to keep track of this at each stage, and it will need to be permanently stored away in case the micro has to 'go off again'. We will use a zero page location, labelled INDIC, for this. We will set bit seven of INDIC to one if the last operation was a deposit, and set it to zero if it was a withdrawal. The reason for using bit 7 is that it is easily tested with a BPL or BMI test. So, if a deposit is required to be made but X = Y and (INDIC$_7$) = 1 (i.e. bit 7 of INDIC is one), then the queue is full and an error condition is required. This is most easily done by setting carry before exitting. The micro will now inspect carry and take appropriate action to safeguard the data. Similarly if a withdrawal is demanded, but X = Y and (INDIC$_7$) = 0, the queue is empty.

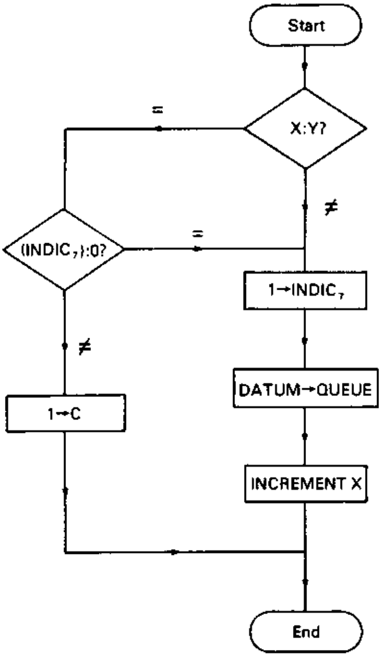


*Figure 6.6: Flow chart for depositing on item in a queue*

Let us consolidate all this with a flowchart (see Appendix 5 for flowcharting conventions). Figure 6.6 shows the flowchart for depositing an item.

LISTING 6.5

```
 10DIM START 50:VDU14
 20 DATUM=&70:INDIC=&71:DIM BASMEM 255
 30DIM TEST 256
 40X%=0:Y%=0:?INDIC=0
 50FOR I%=0 TO 2 STEP2:P%=START
 60[OPTI%
 70STX MEMLOC+1
 80.MEMLOC
 90CPY #0 (Dummy operand)
100BNE OK
110LDA INDIC
120BPL OK
130SEC
140RTS
150.OK
160LDA #&80
170STA INDIC
180LDA DATUM
190STA BASMEM,X
200INX
210CLC
220 RTS:]NEXTI%
230FOR I%=0 TO 256
240?DATUM=RND(256)-1:?(TEST+I%)=?DATUM
250!&404=USR(START):X%=?&405:Y%=?&406
260IF (?&407 AND 1) =1 THEN PRINT"ERROR AT
    "STR$(I%+1)"TH DEPOSIT":GOTO280
270 NEXTI%
280PRINT'"        DEPOSIT   QUEUE"'
290FOR I%=0 TO 255
300PRINT ?(BASMEM+I%),?(TEST+I%):NEXTI%
310VDU15
```

Listing 6.5 shows the program to deposit an item in the queue. Let us concentrate first on the assembly listing (lines 70 to 220). The table below gives the necessary documentation:

| *Line(s)* | *Comment* |
|---|---|
| 70-90 | Device to compare X and Y registers |
| 100 | If not equal, no need to check last transaction |
| 110-120 | Otherwise, check last transaction and if a withdrawal, OK |
| 130-140 | If deposit, indicate error condition and return |
| 160-170 | Store present transaction (a deposit) in indicator |
| 180-190 | Put datum in next free space in queue; |
| 200 | and update pointer to next free space |
| 210-220 | Set non-error condition and return. |

Before going on, it is worth making a few points about this style of documentation. Firstly, it is possible to document the program direct (line 90 is an example), but it is very wasteful of memory and tends to make the listing very cluttered unless TAB's are used. It is better, therefore, to keep documentation separate. The only exception is as line 90, where an instruction is not what it seems - then it is essential to include documentation in the program itself. Secondly, documentation is essential, but over-documentation is an obfuscation and is unfortunately much too common. Over-documentation means stating the obvious. A typical example would be the following:

| | |
|---|---|
| 70 | Store X in the memory location MEMLOC + 1 |
| 80 | Label for temporary storage |
| 90 | Compare Y to the values of X stored in this location |

```
100         If they are not equal branch to OK
110-120     Otherwise set carry and return
            etc.
```

There is no merit in describing the meaning of the instructions: it is their *purpose* which is required when documenting. Sometimes the line between these styles is thin, but it is nevertheless important to keep on the right side of it.

Thirdly, a few words about flowcharting. Flowcharts are useful for small, self-enclosed algorithms, but become confusing and less useful for organising larger programs. One approach is to try to split the program into small parts, and flowchart each part, but this is not always feasible. Don't feel that every program you write has to be flowcharted: flowchart when and if you think it will be useful. And when you flowchart, do not try to pack too much into it.

Let us return to the rest of the program in Listing 6.5, since there are a number of new ideas in it. The purpose of the program surrounding the assembly code is to test the working of that code. The code would never be *used* in a BASIC program like this: it would be part of a larger servicing routine, in all probability. Nevertheless, it has to be tested, and the approach used here is a simple way to do this.

There are a number of points to be made. Firstly, in line 40 we set X% and Y% to zero. The reason for this is that when a machine code routine is called from BASIC the X and Y registers are set to the least significant bytes of the variables X% and Y%. Usually, this is irrelevant to us since we set X and Y in the machine code program itself, but in this case, where we are testing a fragment of a program, X and Y are not set inside the program and so must be initialised before entering it. Note also that the least significant byte of A% is put into the accumulator on entry, though we do not need to exploit this here. Similarly, we set INDIC to represent a withdrawal, so that on first entry when X = Y, we do not get an error.

In line 240, we set DATUM to a random byte, and store it in a 'TEST area' for later recall. We then enter the machine code program in line 250 with the USR function. We recall from Section 5.7 that the value returned consists of four bytes, PYXA, most significant to least significant, where P is the processor status register. By putting USR equal to !&404, we ensure that the least signficant byte of A% equals the value of the accumulator on exit (the permanent address of A% is &404 - &407). This is useful if we wish to return this value to the accumulator on the next entry to the machine code program, and although we do not require this for our present program, it is useful to follow this convention.
In 250 we set X% and Y% to their values on leaving the machine code routine so that when re-entered the values of the X and Y registers are correctly updated.

Line 260 checks whether or not carry is set. Running the program shows that the queue correctly fills to the 256th deposit, but that on the 257th, an error condition is recorded since the queue is full.

### Exercise 6.3

1. Draw the flowchart for the withdrawal of an item of data. Write a program to do this and test it in the same way as in Listing 6.5. Fill the queue with 256 random bytes (as in Listing 6.1). Put the withdrawals in a separate TEST area for later comparison, and do this by having the withdrawal in the accumulator on exit from the routine.

2. Combine the two routines into a single program (keeping the routines separate) which will accept a byte from the keyboard using BASIC and will go to the deposit routine (say at START1) if it is positive (i.e. bit 7 = 0), otherwise go to the withdrawal routine (at START2). Start with the queue half-full (so that X% = 128, Y% = 0). At each stage print out the length of the queue, the item at its head and the item at its tail.

## 6.6 THE ASSEMBLER EQUIVALENT OF PRINT

The writers of the BBC Computer's operating system have included a special subroutine for printing out characters to the screen. It is usually referred to as OSWRCH (for Operating System Write Character), and its address is &FFEE. It will print out the single character whose ASCII code is in the accumulator.

We will discuss subroutines at length in Chapter 9, but for now we need only know that a subroutine is called with the instruction JSR (jump to subroutine) and that at the end of the subroutine an RTS instruction must be placed (return from subroutine). The idea is identical to GOSUB in BASIC: on return, the instruction directly following the JSR instruction is obeyed. As we will see in Chapter 9, sometimes we wish to write our own subroutines, but there are certain very basic subroutines, like OSWRCH, which every computer manufacturer will include, so we would never need to write these ourselves.

You have probably met the idea of ASCII already. In BASIC, we can print out a character with the ASCII code CODE with the instruction PRINT CHR$(CODE). The assembly language equivalent of this is LDA #CODE: JSR OSWRCH (where OSWRCH has already been defined as &FFEE in the program). ASCII stands for American Standard Code for Information Interchange, and its function is to provide an agreed set of codes for every symbol commonly needed in computer communication. It was originally developed for exchange between a terminal and a computer, and for this reason the most significant bit was assigned a special function: it is called the parity bit, and it serves to check that the other 7 bits have been received correctly. Using odd parity, this bit is set to zero or one accordingly, so as to make the sum of the bits an odd number. There are therefore 128 codes available (from 0 to &7F) in ASCII. However, when used internally (as in the BBC computer) no parity check is needed and so there are a full 256 codes available to the user; only the first 128 are accessible from the keyboard, though, these being the standard ASCII set.

It is, fortunately, not necessary to learn these codes, or indeed to have of a table of them, since the BBC computer will give any ASCII code; by typing PRINT ASC ("A"), for example, the number 65 will be returned, the ASCII code for A. The only symbols where one cannot do this are: carriage return (13), quote (34), delete (127) and escape (27). In any case, if one requires to print out text, no reference to ASCII needs to be made by the programmer. Let us see how to print out some text.

All text should be put at the end of a program. To do this make the last statement the label .TEXT. Now when outside the assembly program, the text can be defined by writing $TEXT = "This is an example". The characters of this text will be encoded in ASCII, and the first will be put at the memory location TEXT, which we have defined as the first free location after the program. The very last character will be a carriage return (ASCII code 13). Don't forget to allow for the text when allocating space using the DIM statement.

Now in order to print out this text, the following small routine is required:

```
OUTPUT1 LDA TEXT, X
        CMP #13
        BEQ ENDTEXT
        JSR OSWRCH
        INX
        BNE OUTPUT1
ENDTEXT RTS
```

This will print out the text, without the carriage return. The BNE LOOP instruction will always result in a branch as long as the text does not exceed 255 characters, and saves space over the JMP as well as being intrinsically relocatable. Call it as a subroutine by writing LDX #0 : JSR OUTPUT1 (saving the old value of X beforehand, if necessary).

If we require to print out the carriage return also, we have to take a little care. Carriage return on its own does not generate a new line: it only returns the cursor to the beginning of the current line. To get a new line we must also output code 10, which moves the cursor down one line. Thus to get a new line we write:

```
LDA #10
JSR OSWRCH
```

```
LDA #13
JSR OSWRCH
```

This is needed so often that the designers have included it as a separate subroutine, called OSNEWL at &FFE7. Hence to get a new line we need only write JSR OSNEWL (with OSNEWL defined at the beginning of the program, of course).

It would be useful if we had a subroutine which automatically jumped to OSNEWL if it encountered 13 in the accumulator, otherwise jumping to OSWRCH. There is such a subroutine and it is called OSASCI (at &FFE3). Hence, to print out text with a new line at the end, we use this routine:

```
OUTPUT2 LDA TEXT, X
        JSR OSASCI
        INX
        CMP #13
        BNE OUTPUT2
        RTS
```

Call with LDX #0 : JSR OUTPUT2.

You may be wondering how we accommodate generally a whole series of print statements in a single program. This is a little more complicated and will be covered in full in Chapter 10, although 6.7 deals with two print statements in a less general way.

Finally, a word should be said about the VDU function in BASIC. You will have certainly used this function, and you will be pleased to know it is available in assembler too. Actually this puts things the wrong way round: all that VDU does is to send a byte to OSWRCH, so that it is primarily an assembly code statement. You know that VDU 12 will clear the screen. Hence writing LDA #12 : JSR OSWRCH will clear the screen. Similarly LDA #2 : JSR OSWRCH will turn on the printer. In ASCII, codes 0 to 31 are reserved as control codes: the BBC computer gives these codes special functions which are specified in the VDU Driver section of your manual.

Moreover, this opens up all the colour and graphics facilities in a very simple way. COLOUR 2: VDU 19, 2, 9, 0, 0, 0 sets logical colour 2 to actual colour 9 (flashing red-cyan). In assembler we write:

```
LDA #17
JSR OSWRCH
LDA #2
JSR OSWRCH
LDA #19
JSR OSWRCH
```

```
LDA #2
JSR OSWRCH
LDA #9
JSR OSWRCH
LDA #0
JSR OSWRCH
JSR OSWRCH
JSR OSWRCH
```

Again, PLOT 85, X, Y will draw a triangle from the two points last named to (X, Y). In assembler we have:

```
LDA #85
JSR OSWRCH
LDA # X MOD256
JSR OSWRCH
LDA # X DIV256
JSR OSWRCH
LDA # Y MOD256
JSR OSWRCH
LDA # Y DIV256
JSR OSWRCH
```

X and Y are sent as two byte numbers, lower byte first in each case. The BASIC functions MOD and DIV make this easy to write in assembly language too.

And this means that it is not necessary, nor indeed is it desirable, to write complicated graphics handling programs in assembly language. We can maintain easy portability between languages and processors and make life easy for ourselves by using the same colour and graphics functions that we have in BASIC.

Nevertheless, you will have noticed that just a few VDU statements produce a huge number of assembly instructions (at least 2 and sometimes 4 for each number in the VDU statement). This can become very tedious to type in. In Section 6.8 we will consider ways round this problem.

Exercise 6.4

Using OUTPUT1 and OUTPUT2 as subroutines, write programs to perform in assembly language the following BASIC statements:

1. PRINT "This is question one"

2. PRINT "This is question two";

3. PRINT

4. PRINT "This is question  "; : PRINT "four"

## 6.7 THE ASSEMBLER EQUIVALENT OF GET$, INKEY$ and INPUT A$

A familiar construction in BASIC is:

```
10    A$ = GET$: IF A$ <> " " THEN 10
```

This waits until the space key is pressed before continuing. This is easy to do in assembler, using the operating system read character routine, OSRDCH (at &FFEO). This performs exactly the same function as GET$, putting the ASCII value of the character into the accumulator. So, here we have:

```
LOOP JSR OSRDCH
     CMP #ASC(" ")
     BNE LOOP
```

There is one problem with this: if the escape key is pressed, the system will crash! To stop this happening, we must acknowledge escape using OSBYTE (at &FFF4), a general purpose subroutine, putting &7E into the accumulator. If we wish to otherwise ignore the escape we can write:

```
 LOOP JSR OSRDCH
      CMP #&1B    (the ASCII code for ESC)
      BNE NTESC
      LDA #&7E
      JSR OSBYTE
NTESC CMP #ASC(" ")
      BNE LOOP
```

Clearly, other responses to escape can be accommodated by appropriate coding after returning from OSBYTE.

Another construction is INKEY$. In BASIC we may have 10 ANSWER$ = INKEY$(500). This waits for 5 seconds before continuing, unless a key is pressed. This can be achieved by using OSBYTE again, with &82 in the accumulator. The time is contained in the X and Y registers (X the low-byte); otherwise the function is as GET$ (and escape must be acknowledged in the same way). Here, since 500 =

256 + 244, we write:

```
                                LDX #244
                                LDY #1
                                LDA #&82
                                JSR OSBYTE
```

On return, if no key has been pressed the accumulator will contain &80, and suitable action can be taken.

When using GET$ and INKEY$ in BASIC you will probably have used the statement *FX15, 1 to make sure the keyboard buffer is empty before the keyboard is inspected: this is termed flushing the keyboard buffer. This is easily achieved in assembler using OSBYTE again, this time with 15 in the accumulator and one in the X register.

Thus we write:

```
                                LDX #1
                                LDA #15
                                JSR OSBYTE
```

In fact, all *FX commands are just the operating system mnemonics for OSBYTE. Hence, *FX5, 1 (which selects the Centronics printer port) is equivalent to putting 5 in the accumulator, 1 in the X register and jumping to the OSBYTE routine. It follows that any *FX command is easily translated into assembly code.

Finally 10 INPUT A$. This can be achieved by writing a subroutine using just GET$, but a system call exists which does everything required. This is OSWORD (at &FFF1), with zero in the accumulator. Unfortunately, its usage is a little complicated. It is far easier to jump to part of the BASIC interpreter to perform this task. JSR &BC20 will read text into the direct command input buffer beginning at &0700. Pressing escape will automatically return the computer to direct command status (i.e. out of the program) and pressing CTRLU will erase the entire line of text input. The text will end with carriage return (code 13). Here is an assembly program to produce the equivalent of the following BASIC statement:

```
      10     INPUT A$: IF LEFT$(A$,2) = "NO" THEN 100 ELSE IF
      LEFT$(A$,3) <> "YES" THEN 10
                          START JSR &BC20
                                LDA &700
                                CMP #ASC("N")
                                BNE NTNO
                                LDA &701
```

```
                CMP #ASC("O")
                BEQ NO
         NTNO   LDA &700
                CMP #ASC("Y")
                BNE START
                LDA &701
                CMP #ASC("E")
                BNE START
                LDA &702
                CMP #ASC("S")
                BNE START
         YES        .
                    .
                    .
                    .
                    .
         NO         .
                    .
                    .
                    .
```

It would not be difficult to combine this with the ideas of the last section to produce a more friendly computer response. For example, in BASIC:

```
10      INPUT "PLEASE ANSWER YES OR NO" A$: IF LEFT$(A$,2) =
"NO" THEN 100 ELSE IF LEFT$(A$,3) <> "YES" THEN PRINT
"I DON'T UNDERSTAND YOUR ANSWER": GOTO 10
```

As an exercise encode this into assembler, type it into the computer and see if it works. Put the statements to be printed into $TEXT and $(TEXT + 24) respectively (since the first statement is 24 characters long including the carriage return). Print out the first statement by using LDX #0 : JSR OUTPUT1 and the second with LDX #24 : JSR OUTPUT1. Produce a new line with JSR OSNEWL.

If we wish to compare text to a longer string than "YES", then the above method is a little clumsy. Listing 6.6 is a program which will accept input from the keyboard and test whether it lies alphabetically between two strings. The two strings are input into the BASIC part of the program before the machine code routine is called, and put into zero page. Listing 6.6 contains the details.

Lines 60 and 70 print out a question mark prompt. Lines 110 to 150 compare the input string to the bottom string limit. Exit from this loop to line 160 will only occur if the input is higher alphabetically than the bottom limit. Thus if the bottom limit is MISTY

LISTING 6.6

```
   10OSWRCH=&FFEE
   20DIM START 50
   30FOR I%=0 TO 2 STEP2
   40P%=START
   50[OPTI%
   60LDA #ASC("?")
   70JSR OSWRCH
   80JSR &BC20
   90LDX #&FF
  100.LOOP1
  110INX
  120LDA &700,X
  130CMP &70,X
  140BCC OUTSIDE
  150BEQ LOOP1
  160LDX #&FF
  170.LOOP2
  180INX
  190LDA &80,X
  200CMP &700,X
  210BCC OUTSIDE
  220BEQ LOOP2
  230CLC
  240RTS
  250.OUTSIDE
  260SEC
  270RTS:]NEXTI%
  280INPUT"Lower string limit",A$:IF LEN(A$)>15 THEN280
  290INPUT"Upper string limit",B$:IF LEN(B$)>15 THEN290
     ELSE IF A$>B$ THEN280
  300PROCASC(A$,&70)
  310PROCASC(B$,&80)
  320PRINT"Input your string":!&404=USR(START)
  330IF (?&407 AND 1)=0 THEN PRINT"Inside" ELSE PRINT
     "Outside"
  340PRINT'"Another?"
  350G$=GET$:IF G$="Y" THEN 320 ELSE IF G$<>"N" THEN 350
  360PRINT"New string limits?"
  370G$=GET$:IF G$="Y" THEN 280 ELSE IF G$<>"N" THEN 370
     ELSE END
  380DEF PROCASC(S$,M%)
  390S$=S$+CHR$(13):M%=M%-1
  400FOR I%=1 TO LEN(S$)
  410?(M%+I%)=ASC(MID$(S$,I%,1))
  420NEXTI%
  430ENDPROC
```

and MIST is input, equality will occur for the first four letters, but then on the fifth we compare 13 (carriage return) to 89 (ASCII value of Y), obtain a negative result and branch to OUTSIDE. Equally, if the bottom limit is MIST and MISTY is input, then we compare 89 to 13, obtain a positive result and go through to line 160. In the same way lines 180 to 220 compare the input string to the upper limit. If the string is inside the limits, C is set to zero, otherwise it is set to 1. This is picked up at line 330 (as in Listing 6.5, line 260), and the appropriate output delivered.

The procedure at lines 380 to 430, called at lines 300 and 310, is necessary since the $TEXT statement will not function if TEXT is in zero page. We use zero page here to save memory and gain speed with the zero page indexed addressing mode (lines 130 and 190).

Sometimes we may wish to use a different input buffer from &700 since &700 is of no use for anything other than temporary storage. Once outside the program all input will be lost. If we wish to permanently store the strings input, it would be more convenient to store them directly. We can do this as follows: put the lower byte of the address of the buffer in &37 and the higher byte in &38 and use JSR &BC28. In addition, we can restrict the length of the string input by loading the accumulator with the length and using JSR &BC2A instead of &BC28. The default length is 238 otherwise.

Listing 6.7 illustrates this with a program which stores a set of strings input in an array. No string is allowed to exceed 19 characters, since the strings are to be stored in fixed widths of 20 locations. In the next chapter, we will be able to relax these restrictions, and use variable width locations.

LISTING 6.7

```
 10OSWRCH=&FFEE
 20 DIM START 500
 30FOR I%=0 TO 2 STEP2
 40P%=START
 50[OPTI%
 60LDA #TEXT MOD 256
 70STA &37
 80LDA #TEXT DIV 256
 90STA &38
100.BEGIN
110LDA #ASC("?")
120JSR OSWRCH
130LDA #19
140JSR &BC2A
150LDA &37
160CLC
170ADC #20
180STA &37
190BCC BEGIN
200INC &38
210BCS BEGIN
220.TEXT:]NEXTI%
230CALL START
```

Line 20 allows storage for at least 23 strings, starting at the location TEXT defined in 220. Lines 60 to 90 store the lower and higher bytes of TEXT in &37 and &38, and line 130 restricts the string input to 19 characters. Lines 150 to 210 increment the storage address by 20 for the next input. As it stands, we can exit from this program only by pressing ESCAPE; we will also improve on this in the next chapter.

Exercise 6.5

1. Write a program which will print out any string input in reverse. Use an infinite loop with exit by ESCAPE only.

2. Write a program which will print out any string input with all spaces removed. Loop until * alone is input, and then end.

3. Write a program to store a string starting in location TEXT (at the end of the program), with only the trailing spaces removed (i.e. remove any  spaces after the last 'visible' character input). Include a suitable test of your program.

## 6.8 MACROS, CONDITIONAL ASSEMBLY AND TABLES: SIMPLIFYING VDU STATEMENTS

Consider the following set of instructions in BASIC:

```
MODE 4
VDU28,0,31,39,16        (set bottom half of screen for text)
VDU24,0;512;1273;1023;  (set top half of screen for graphics)
VDU29,0;512;            (move graphics origin to bottom LH
                         corner of graphics screen)
VDU19,1,0;0;            (foreground black)
VDU19,0,3;0;            (background yellow)
MOVE 0,0
DRAW 450,450
```

The result of all this is to draw a line in the top half of the screen in black on a yellow background. We can in fact write this as one VDU statement, replacing MODE, MOVE and DRAW by their VDU equivalents:

```
VDU22,4,28,0,31,39,16,24,0;512;1273;1023;29,0;512;19,1,0;0;
19,0,3;0;25,4,0;0;25,5,450;450;
```

To encode this into assembly language would require us to type in 90 or so lines of code (there are 31 numbers, but 14 are double bytes, being followed by a semi-colon e.g. 0; is &0000). There must be an easier way! Indeed, there are two such ways.

LISTING 6.8

```
  10OSWRCH=&FFEE
  20DIM START 500
  30FOR I%=0 TO 3 STEP 3:P%=START:RESTORE
  40[OPTI%
  50LDA #0
  60STA &70
  70STA &71
  80]PROCVDU(39)
  90DATA 22,4,28,0,31,39,16,24,0,0,512,1273,1023,
        29,0,0,512,19,1,0,0,0,0,19,0,3 ,0,0,0,25,
        4,0,0,0,0,25,5,450,450
 100[OPTI%
 110RTS:]NEXTI%
 120CALLSTART:END
 130DEF PROCVDU(N):LOCAL D,D$,H,L,J%
 140FOR J%=1 TO N:READ D$:D=EVAL(D$)
 150IF D>255 THEN H=D DIV 256: D= D MOD256 ELSE H=-1
 160IF D<>L THEN [OPTI%:LDA #D:]
 170[OPTI%:JSR OSWRCH:]
 180L=D:IF H=-1 THEN190 ELSE D=H:H=-1:GOTO160
 190NEXTJ%
 200ENDPROC
```

Listing 6.8 shows the first way. Lines 50-70 are included solely to show how the VDU statement can be put into the middle of an assembly program: they are quite arbitrary. Lines 80 and 90 contain the new ideas. Line 80 begins with an 'end assembly code' marker (]), and then a procedure is called. Line 90 contains the VDU numbers, with the small difference that 0; has been replaced by 0,0 and 3; by 3,0.

The procedure is in lines 130-200. Line 130 specifies the variables used as local so that they may be used again if required outside the procedure (the parameter N is automatically local). Line 140 reads the data from line 90 into the variable D. (It is read first into D$ and EVAL applied to it to accommodate hex numbers e.g. &1000 could be an item of data.) Line 150 checks whether D exceeds 255: if it does it splits it into two bytes, with H the higher and D the lower; otherwise it sets H to -1 as a flag.

Line 160 is an example of what is called *conditional assembly.* L contains the last byte output. If D is equal to L then we do not need a LDA statement because the accumulator will already contain this value. If D is not equal to L, the LDA statement is required. Notice the inclusion of OPTI% here: when we re-enter assembly mode (by using the marker [) OPT will be set to its default value 3 unless we re-specify it. I% is, of course, the current OPT value from the beginning of the program (i.e. 0 or 3).

Again in 170 OPTI% is needed. Line 180 will set L (the last byte output) to D, and then will either read the next byte from the data, or if H contains a value will transfer that to D instead.

The parameter N contains the number of items in the data statement. Finally note the importance of RESTORE in line 30: there are two passes and the DATA statement must be restored after the first one.

If you run this with the page mode on (CTRLN) you will see all the LDA #D, JSR OSWRCH statements being assembled. Normally you would use it using I% = 0 TO 2, but we use I% = 3 here to demonstrate that all the VDU statements have been translated into assembly code.

This method of producing code is called *macro-assembly* and the procedure is an example of a *macro*. Macro-assemblers are usually only found on large expensive machines, but the facility is available to use because BASIC and assembly code can be mixed.

LISTING 6.9

```
 10OSWRCH=&FFEE
 20DIM START 500
 30FOR I%=0 TO 3 STEP 3:P%=START:RESTORE
 40[OPTI%
 50LDA #0
 60STA &70
 70STA &71
 80LDX #0
 90.LOOPVDU
100LDA TABLE,X
110JSR OSWRCH
120INX
130CPX #45
140BNE LOOPVDU
150BEQ OVERTABLE
160.TABLE
170]PROCTABLE(39)
180DATA 22,4,28,0,31,39,16,24,0,0,512,1273,1023,29,
        0,0,512,19,1,0,0,0,0,19,0,3 ,0,0,0,25,4,0,0,
        0,0,25,5,450,450
190[OPTI%
200.OVERTABLE
210RTS:]NEXTI%
220CALLSTART:END
230DEF PROCTABLE(N):LOCAL D,D$,H,J%
240FOR J%=1 TO N:READ D$:D=EVAL(D$)
250IF D>255 THEN H=D DIV 256: D= D MOD256 ELSE H=-1
260?P%=D: P%=P%+1
270IF H=-1 THEN 280 ELSE D=H:H=-1: GOTO260
280NEXTJ%
290ENDPROC
```

There is a second way of coding the VDU statement, and Listing 6.9 shows it. Again lines 50-70 are arbitrary. Lines 80-140 perform a loop to load the accumulator with the next item of data from the table (stored at the address TABLE defined in line 160) and output it with OSWRCH. 45 is used as the comparison since there are 45 bytes (31 numbers but 14 are double bytes).

The table of data values is set up using the procedure called in line 170. Using exactly the same data as in Listing 6.8 (at line 180 here), line 260 puts each item of data into the current address pointed to by the program counter (stored automatically in P%) and increments the counter. When first entered P% = TABLE. On exit, P% will point to the first free location following the 45 bytes stored in the table. The table could have been stored at the end of the machine code program, but it is instructive to see here how it can be put in the middle when required.

Running this with CTRLN set points to the essential difference between the methods employed in Listing 6.8 and Listing 6.9. In the latter, the coding to perform the output to the VDU is written only once, but performed many times. In the former, the coding to perform the output is written many times but performed only once. Thus the macro approach is an assembly time facility, whereas the table approach is a run-time facility.

In general, the table approach is much cheaper on memory than the macro approach, but it is slightly slower in execution time (since the table has to be read, X has to be incremented and compared to the limit, and a branch has to be made). As a rule, then, use tables unless speed is critical, when the macro approach can be used.

## Exercise 6.6

Use a macro with conditional assembly to:

(a) rewrite the coding to Q.2, Ex.3.2

(b) combine the coding for Q.2 Ex.3.2 and Q.2 Ex.3.3 into a single assembly program, an index being set to A if Ex.3.2 is to be produced and to S if Ex.3.3 is to be produced.

In each case suppress the assembler listing.

# *Chapter 7 Indirect indexed addressing*

## 7.1 MOVING A SECTION OF MEMORY

Let us consider how we might move a section of memory which is more than 256 bytes long. In Section 5.4c we saw a general method of performing a for/next loop with more than 256 cycles. We can modify that idea for our present task. Listing 7.1 shows the program.

LISTING 7.1

```
 10NUML=&70:NUMH=&71
 20INPUT"How many bytes will be moved",!NUML
 30INPUT"Starting address of memory to be moved",
   A$:OLDLOC=EVAL(A$)
 40INPUT"Starting address of new location",A$:NEWLOC=EVAL(A$)
 50DIM START 100
 60FOR I%=0 TO 2 STEP 2:P%=START
 70[OPTI%
 80LDX NUMH
 90BEQ LOLOOP
100LDY #0
110.MEMLOC1
120LDA OLDLOC,Y
130.MEMLOC2
140STA NEWLOC,Y
150INY
160BNE MEMLOC1
170INC MEMLOC1+2
180INC MEMLOC2+2
190DEX
200BNE MEMLOC1
210.LOLOOP
220 LDX NUML
230BEQ FINISH
240.LOOP
250LDA OLDLOC+256*?NUMH,Y
260STA NEWLOC+256*?NUMH,Y
270INY
280DEX
290BNE LOOP
300.FINISH
310RTS:]NEXTI%
320CALL START
330FOR I%=0 TO 256*?NUMH+?NUML-1
340IF ?(OLDLOC+I%)<>?(NEWLOC+I%) PRINT "Error at move"I%+1:END
350NEXTI%
```

As in Chapter 6, the memory to be moved starts at OLDLOC and it is to be moved to a section of memory starting at NEWLOC. Altogether, (NUMH; NUML) bytes are to be moved. We move the (NUMH) sets of 256 bytes first. For reasons which will become clear in a moment, we use Y to index the move in each cycle and use X to count the number of sets of 256 cycles. X is initialised in line 80 (and if (NUMH) = 0, an immediate branch is taken to the move of the (NUML) bytes at 220). Then Y is initialised, and the 256 byte move occurs in lines 120-160.

In lines 170 and 180 we increment the high byte of the addresses OLDLOC and NEWLOC before repeating the 256 cycle move. For example, if OLDLOC is &8320 and NEWLOC is &4000, then at the end of the first 256 cycles we have moved the contents of locations &8320-&841F to locations &4000-&40FF. At the beginning of the next 256 cycles we want OLDLOC to begin at &8420 and NEWLOC to begin at &4100, and lines 170 and 180 achieve this. The process continues until we have moved all (NUMH) bits of 256 bytes. Then in 220-290 we move the residual number of bytes expressed in (NUML). Again, continuing the example, suppose we are moving &920 bytes; then (NUML) = &20. In lines 250 and 260 we begin the move from &8D20 (&8420 + 256 * 9), as required, and move it to &4900. The move ends at &491F, giving &920 bytes moved in all.

You may be wondering why in 220-290 we do not simply load Y with the contents of NUML and decrement Y as we go, so dispensing with X altogether. This would not work, however, since we would move a byte from &8D20 to &4920, and this should not happen. To make this work we would have to decrement Y before starting the move (so that Y contained (NUML) - 1), and check each stage of the loop with a CPY #&FF, since Y = 0 must be included. It turns out that all of this takes up more space and more time than using both the X and Y registers.

The reason we have used Y to index each cycle of the move rather than X is to accommodate the case where either OLDLOC or NEWLOC are zero page locations. If we used X in this case, the high byte of the address would not be available for incrementing (see Section 6.3 to remind yourself of this). Using Y, zero page addresses are stored in two bytes e.g. if OLDLOC is &10,

```
          LDA OLDLOC, X is translated as B5 10   whereas
          LDA OLDLOC, Y is B9 10 00.
```

There is one fundamental problem with this program. It works well, but it needs to be reassembled every time we wish to change the number of bytes to be moved or the starting addresses OLDLOC or NEWLOC. This means that the program could never be a stand-alone machine code program; it would always have to be tied to the assembler. This is not good programming practice: in general, we should only have to assemble a program once; thereafter, any change in data should not require reassembly.

A solution to this problem is shown in Listing 7.2. The address OLDLOC is put into two bytes OLDLOCL and OLDLOCH and passed as data. Similarly the address NEWLOC is put in NEWLOCL and NEWLOCH.

LISTING 7.2

```
 10NUML=&70:NUMH=&71:OLDLOCL=&72:OLDLOCH=&73:NEWLOCL=&74:NEWLOCH=&75
 20INPUT"How many bytes will be moved",!NUML
 30INPUT"Starting address of memory to be moved",A$:!OLDLOCL=EVAL(A$)
 40INPUT"Starting address of new location",A$:!NEWLOCL=EVAL(A$)
 50DIM START 100
 60FOR I%=0 TO 2 STEP 2:P%=START
 70[OPTI%
 80LDA OLDLOCL
 90STA MEMLOC1+1
100STA MEMLOC3+1
110LDA OLDLOCH
120STA MEMLOC1+2
130STA MEMLOC3+2
140LDA NEWLOCL
150STA MEMLOC2+1
160STA MEMLOC4+1
170LDA NEWLOCH
180STA MEMLOC2+2
190STA MEMLOC4+2
200LDX NUMH
210BEQ LOLOOP
220LDY #0
230.MEMLOC1
240LDA &FFFF,Y Dummy operand
250.MEMLOC2
260STA &FFFF,Y Dummy operand
270INY
280BNE MEMLOC1
290INC MEMLOC1+2
300INC MEMLOC2+2
310INC MEMLOC3+2
320INC MEMLOC4+2
330DEX
340BNE MEMLOC1
350.LOLOOP
360 LDX NUML
370BEQ FINISH
380.MEMLOC3
390LDA &FFFF,Y Dummy operand
400.MEMLOC4
410STA &FFFF,Y Dummy operand
420INY
430DEX
440BNE MEMLOC3
450.FINISH
460RTS:]NEXTI%
470CALL START
480FOR I%=0 TO 256*?NUMH+?NUML-1
490IF ?(!OLDLOCL MOD65536+I%)<>?(!NEWLOCL+I%) PRINT "Error at move"I%
500NEXTI%
```

Lines 80-190 take this data and store it into the relevant parts of the program. Thus the address OLDLOC is put into lines 240 and 390; and the address NEWLOC into lines 260 and 410. The program then operates identically to Listing 7.1, except that the addresses in 390 and 410 are also incremented. Once this program has been assembled, we can reuse it with any new data without reassembly.

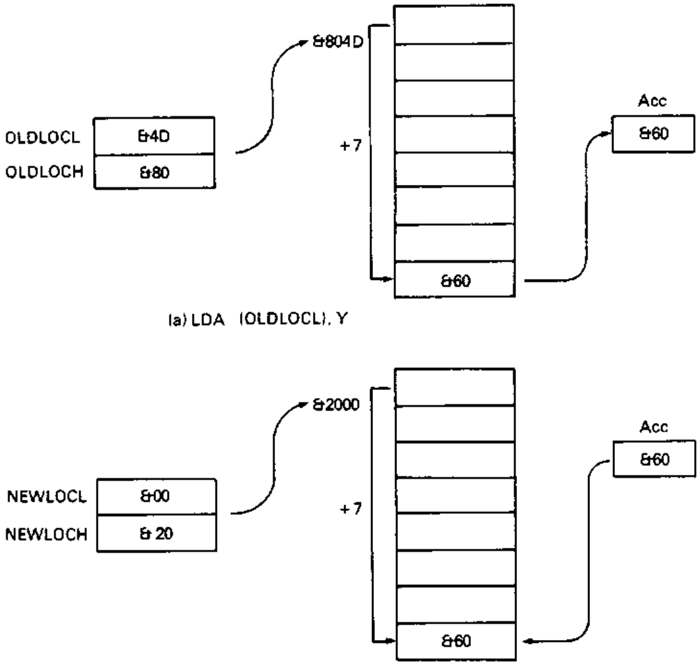
## 7.2 A BETTER METHOD

The solution given in Listing 7.2 works but it is clumsy and long-winded. Moreover, we could not use it if we wanted to put the program into a chip in ROM. The designers of the 6502 microprocessor have provided us with a better way of solving the problem.

LISTING 7.3

```
 10CLS
 20NUML=&70:NUMH=&71:OLDLOCL=&72:OLDLOCH=&73:NEWLOCL=&74:NEWLOCH=&75
 30DIM START 100
 40FOR I%=0 TO 2 STEP 2:P%=START
 50[OPTI%
 60LDX NUMH
 70BEQ LOLOOP
 80LDY #0
 90.LOOP1
100LDA (OLDLOCL),Y
110STA (NEWLOCL),Y
120INY
130BNE LOOP1
140INC OLDLOCH
150INC NEWLOCH
160DEX
170BNE LOOP1
180.LOLOOP
190 LDX NUML
200BEQ FINISH
210.LOOP2
220LDA (OLDLOCL),Y
230STA (NEWLOCL),Y
240INY
250DEX
260BNE LOOP2
270.FINISH
280RTS:]NEXTI%
290INPUT"How many bytes will be moved",!NUML
300INPUT"Starting address of memory to be moved",A$:!OLDLOCL=EVAL(A$
310INPUT"Starting address of new location",B$:!NEWLOCL=EVAL(B$)
320CALL START:PRINT"Memory moved. Checking now."
330A=EVAL(A$):B=EVAL(B$)
340FOR I%=0 TO 256*?NUMH+?NUML-1
350IF ?(A+I%)<>?(B+I%) PRINT "Error at move"I%+1:END
360NEXTI%
370PRINT"Check OK":GOTO290
```

The program is shown in Listing 7.3, and the new ideas are in lines 100, 110, 220 and 230. LDA (OLDLOCL), Y means exactly the same as line 390 in Listing 7.2 after the new contents have been assigned in lines 80-190. It means load the accumulator with the contents of the location (OLDLOCH; OLDLOCL) + Y. Similarly STA (OLDLOCL), Y stores the accumulator into memory location (OLDLOCH; OLDLOCL) + Y. Figure 7.1 illustrates this.



(a) LDA (OLDLOCL), Y

(b) STA (NEWLOCL), Y

*Figure 7.1: An illustration of the indirect indexed addressing mode, where OLDLOC is &804D, NEWLOC is &2000, Y equals 7 and &8054 contains &60*

This mode of addressing is called *indirect indexed* addressing, and it is very powerful. In Listing 7.3, we are able to increment the addresses pointed to in lines 100 and 220 in one go by writing INC OLDLOCH in line 140. In Listing 7.2 we had to do this twice, once for line 240 (at line 290) and once for line 390 (at line 310).

The mnemonic form used is actually very close to the one introduced in Section 2.7 for our own use. Using that convention, we would be inclined to write LDA (OLDLOCH; OLDLOCL), Y. The 6502 can shorten this because it assumes that OLDLOCH is *always the next location up* from OLDLOCL i.e. OLDLOCH = OLDLOCL + 1. So it can write LDA (OLDLOCL), Y for short, since given OLDLOCL it knows OLDLOCH.

The indirect indexed addressing mode is *always a two byte instruction* because the operand, here OLDLOCL, *has to be a zero page address*. In practice, this is not a serious limitation, and it results in an increase in processor speed and a decrease in memory space used to store the program.

We cannot use the indirect indexed addressing mode with the X register: *it is only available with the Y register*. The X register has a special indirect mode of its own which we shall meet in Appendix 3. Moreover, the indirect indexed mode is not available with all instructions, as a glance at Appendix 1 will demonstrate. It can only be used with ADC, AND, CMP, EOR, LDA, ORA, SBC and STA. But even with these restrictions, a great deal can be done with it as this and later chapters will show.

Since the high and low bytes of the address referenced indirectly in the indirect indexed mode must be stored in touching bytes, it is usual to dispense with the L and H suffixes on the labels. Thus instead of OLDLOCL we write simply OLDLOC; and if we wish to refer to OLDLOCH we do so by using OLDLOC + 1. This convention is illustrated in Listing 7.4, which purports to be an improvement on Listing 7.3. Notice that in line 20 NUM is put equal to &70, OLDLOC to &72 and NEWLOC to &74. This leaves &71 for NUM + 1, &73 for OLDLOCH and &75 for NEWLOC + 1.

LISTING 7.4

```
 10CLS
 20NUM=&70:OLDLOC=&72:NEWLOC=&74
 30DIM START 100
 40FOR I%=0 TO 2 STEP 2:P%=START
 50[OPTI%
 60LDX NUM+1
 70LDY #0
 80.LOOP
 90LDA (OLDLOC),Y
100STA (NEWLOC),Y
110DEY
120BNE LOOP
130INC OLDLOC+1
140INC NEWLOC+1
150DEX
160BMI FINISH
170BNE LOOP
180LDY NUM
190BNE LOOP
200.FINISH
210RTS:]NEXTI%
220INPUT"How many bytes will be moved",!NUM
230INPUT"Starting address of memory to be moved",A$:!OLDLOC=EVAL(A$
240INPUT"Starting address of new location",B$:!NEWLOC=EVAL(B$)
250CALL START:PRINT"Memory moved. Checking now."
260A=EVAL(A$):B=EVAL(B$)
270FOR I%=0 TO 256*?(NUM+1)+?NUM-1
280IF ?(A+I%)<>?(B+I%) PRINT "Error at move"I%+1:END
290NEXTI%
300PRINT"Check OK":GOTO220
```

It is also conventional *in discussion* to refer to all the bytes associated with each label by just stating the label. Thus if we want to refer to OLDLOC and OLDLOC + 1 at the same time, and there is no ambiguity, we just say OLDLOC. Similarly, if a location NUMBER has associated with it four bytes, NUMBER, NUMBER + 1, NUMBER + 2, NUMBER + 3 then we could refer to these by just saying NUMBER. So, for example, the statements 'OLDLOC contains the address of the first location to be moved' and 'NUMBER contains the 32 bit signed integer' utilise this convention and are clearly unambiguous. In the rest of this book, we shall adopt this convention in our discussions where there is no danger of ambiguity.

Example 7.1

1. What relationship must hold between !OLDLOC and !NEWLOC for the program in Listing 7.3 to work?

2. Write a program which will work in those cases where Listing 7.3 does not. Use exactly the same data input as Listing 7.3 (i.e. the start of memory to be moved, the start of the new locations and the number of bytes to be moved). Which program is more efficient?

3. Listing 7.4 purports to be an improvement on Listing 7.3. However, there are some faults in it. Correct what you can, and state which faults cannot be corrected.

## 7.3 INPUTTING A SERIES OF STRINGS OF VARYING LENGTHS

In Chapter 6, Listing 6.7, we constructed a program to input a series of strings into an array. We had to restrict the length of strings to 19, and regardless of the length input, 20 bytes had to be reserved for each string. Moreover, the only way we could indicate that we had finished input was by pressing the ESCAPE key which necessarily took us out of the program.

LISTING 7.5

```
   10OSWRCH=&FFEE
   20DIM START 4000
   30FOR I%=0 TO 2 STEP 2
   40P%=START
   50[OPTI%
   60LDA #TEXT MOD 256
   70STA &37
   80LDA #TEXT DIV 256
   90STA &38
  100LDA #12
  110JSR OSWRCH
  120.BEGIN
  130LDA #ASC("?")
  140JSR OSWRCH
  150JSR &BC28
  160 LDY #0
```

```
170 LDA (&37),Y
180CMP #ASC("*")
190BNE LOOP
200INY
210LDA (&37),Y
220CMP #&0D
230BNE LONG
240RTS
250.LONG
260LDY #0
270.LOOP
280LDA (&37),Y
290CMP #&0D
300BEQ ENDSTRING
310INY
320BNE LOOP
330.ENDSTRING
340INY
350TYA
360CLC
370ADC &37
380STA &37
390BCC BEGIN
400INC &38
410BCS BEGIN
420.TEXT:]NEXTI%
430CALLSTART
440STRING=TEXT:VDU14:REPEAT
450 A$=$STRING :PRINTA$:STRING=STRING+LEN(A$)+1
460UNTIL A$="*"
470VDU15
```

Armed with our new addressing mode, we can improve on all of this. Listing 7.5 allows a series of strings of any length to be input, stored in an array at the end of the machine code. Exit is achieved by the input of a single asterisk; **, for example, will not be interpreted as an exit signal. The details are as follows:

60-90 The low byte of the first free location at the end of the program (referenced by TEXT) is put in &37 and the high byte in &38. This is required for the routine at &BC28 (see Section 6.7).

100-110 Clear the screen.

130-150 Output the ? prompt, and accept input from the keyboard.

160-190 Check if the first character is an asterisk. If not, go to line 280.

200-260 Check if the second character is a carriage return. If so, the single asterisk exit signal has been input and so we return at 240. Otherwise reset the character index pointer (Y) to zero at 260.

280-300 Get the next character. If it is a carriage return, the end of string has been found, so go to 340.

310-320 Otherwise increment the index pointer and branch back to 280. The branch always occurs since the string length cannot reach 256.

340-380 Increment the base address of the buffer for input to the next free location in memory by adding the string length (Y + 1) to the old base address.

390-410 If there is no carry branch immediately to 130 for the next string; otherwise, increment the high byte of the buffer address before branching to 130.

As usual, we test the program using BASIC. Here, on exit from the machine code we print out the ASCII contents of the memory from TEXT onwards until a single asterisk is found.

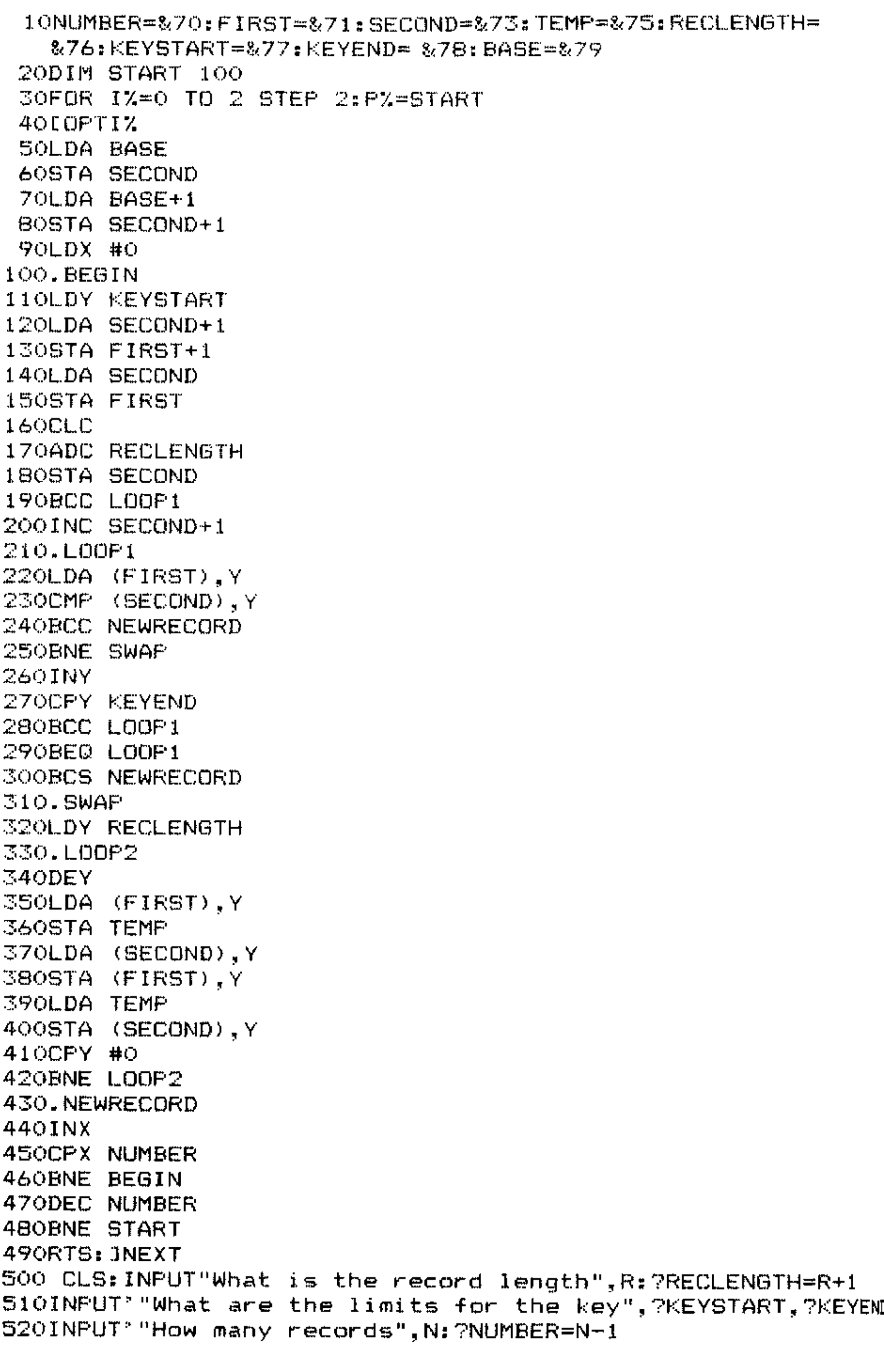
Notice how the indirect indexed addressing mode solves our problems for us. We are able to accommodate variable length strings economically, because the buffer address can be set to the next free location by the simple expedient of adding the last string length to the last buffer address. We are also able to inspect each character of the string, which we could not do in Listing 6.6, and so we can exit from the program without having to use ESCAPE. This would be important if the present program were part of a larger package, where the input strings were to be used.

## 7.4 SORTING A SERIES OF FIXED LENGTH RECORDS

In some applications, it is practicable to use a storage method for records, where each field in the record is given a fixed amount of storage space. Thus the record is of a fixed length. An example of this might be wages data, where, say, 30 characters are given for the name, 5 for a payroll number, 3 for the hourly rate etc. In such cases, it is usually required that we can sort the records, with respect to any of the fields. Thus we may want to sort according to name, or according to hourly rate or according to any other field we choose. In this case, we have no need to mark the end of the records with a carriage return; rather we will require exact details of the whereabouts of the field upon which we will sort the records.

Listing 7.6 gives a program to achieve this. It will sort up to 256 records, each record a fixed length up to 254 characters. The particular field is accessed by two indexes: one for the beginning of the field and one for the end. The indexes are relative to the beginning of the record: so, for example, the hourly rate above would be accessed with indexes 35 and 37, the payroll number with 30 and 34, and the name with 0 and 29.

LISTING 7.6



```
  10NUMBER=&70:FIRST=&71:SECOND=&73:TEMP=&75:RECLENGTH=
    &76:KEYSTART=&77:KEYEND= &78:BASE=&79
  20DIM START 100
  30FOR I%=0 TO 2 STEP 2:P%=START
  40[OPTI%
  50LDA BASE
  60STA SECOND
  70LDA BASE+1
  80STA SECOND+1
  90LDX #0
 100.BEGIN
 110LDY KEYSTART
 120LDA SECOND+1
 130STA FIRST+1
 140LDA SECOND
 150STA FIRST
 160CLC
 170ADC RECLENGTH
 180STA SECOND
 190BCC LOOP1
 200INC SECOND+1
 210.LOOP1
 220LDA (FIRST),Y
 230CMP (SECOND),Y
 240BCC NEWRECORD
 250BNE SWAP
 260INY
 270CPY KEYEND
 280BCC LOOP1
 290BEQ LOOP1
 300BCS NEWRECORD
 310.SWAP
 320LDY RECLENGTH
 330.LOOP2
 340DEY
 350LDA (FIRST),Y
 360STA TEMP
 370LDA (SECOND),Y
 380STA (FIRST),Y
 390LDA TEMP
 400STA (SECOND),Y
 410CPY #0
 420BNE LOOP2
 430.NEWRECORD
 440INX
 450CPX NUMBER
 460BNE BEGIN
 470DEC NUMBER
 480BNE START
 490RTS:]NEXT
 500 CLS:INPUT"What is the record length",R:?RECLENGTH=R+1
 510INPUT'"What are the limits for the key",?KEYSTART,?KEYEND
 520INPUT'"How many records",N:?NUMBER=N-1
```

```
530DIM B ?(RECLENGTH)*N: !BASE=B
540PRINT'"Setting up strings now"
550 FOR I%=0 TO N-1:FOR J%= 0 TO R-1:?(B+I%*(R+1)+J%)
    =RND(26)+64:NEXTJ%:?(B+I% *(R+1)+J%)=13 :NEXTI%
560PRINT"Sorting now.":CALLSTART:PRINT"Checking."
570FOR I%=0 TO (?RECLENGTH)*(N-2) STEP (?RECLENGTH):IF
   MID$($(B+I%),?(KEYSTART )+1,?KEYEND-?KEYSTART+1)
   >MID$($(B+I%+(?RECLENGTH)),?(KEYSTART)+1,?KEYEND-?KEYSTA
   RT+1) THEN PRINT "ERROR AT"STR$(I%):END
580NEXT:PRINT"O.K.":END
```

We use a bubble sort, as discussed in Chapter 6. The program details are:

50-80 Put the base address (of where the records are stored) initially in locations which will point to the second string in the bubble sort comparison.

90 Set the record count to zero.

110 Set the character index pointer to the lower limit for the field to be sorted upon.

120-150 Put the old value of the pointer to the second string of the current pair into the pointer to the first string of the pair.

160-200 Increment the second string pointer to the next record by adding the record length.

220-230 Compare the relevant field of the first record of the current pair of records being considered in the bubble sort with that of the second record.

240-250 If a character fails to match, either the records are correctly ordered (line 240) or they are not (line 250).

260-300 Otherwise, look at the next character. If we are past the limit for the field (i.e. line 300 is entered), then the strings must be identical and need not be swapped.

310-320 The routine to swap a pair of records. Y contains the record length.

340-420 Swap characters one by one, starting at the end of the respective fields. TEMP is used as an storage intermediary.

440-450 Continue until the current number of records have been examined.

470-480 In that case, decrement the current number by one (the bubble principle) and continue until there are no records left to bubble through.

Notice the use of the two sets of locations FIRST and SECOND here to point to the start of the current pair of records. By continually updating these on each cycle ((SECOND) → FIRST, (SECOND) + (RECLENGTH) → SECOND) we are able easily to move through the entire set of records, pair by pair, using indirect indexed addressing.

Again, we test the program using BASIC; here we set up the requisite number of random strings of the correct length, and after sorting, make sure that they are indeed in the correct order.

### Exercise 7.2

Rewrite the program to deal with more than 256 records. Use LOOPCOUNTH to count the high byte of the loop and X for the low byte. Take particular care with the comparison: the problem is probably harder than you think.

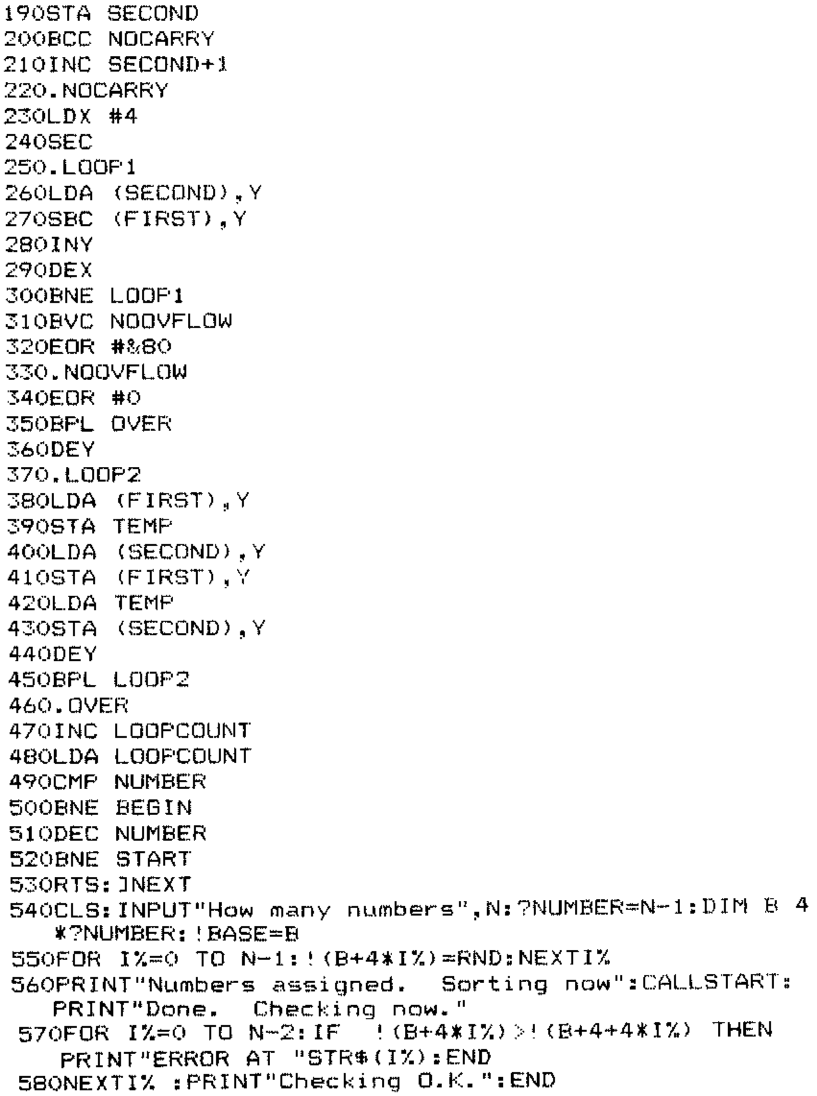
## 7.5 SORTING A SERIES OF 32 BIT SIGNED INTEGERS

In Section 6.4 we saw how to sort a series of bytes into numerical order. Using indirect indexed addressing we are now able to go much further than this: we will write a program to sort a series of 32 bit integers into numerical order. You will recall that all integers in BASIC are stored in this format, so we will have a program which could sort BASIC integer variables (see 10.2 for details of this).

Once again we will use the bubble sort. This is a fairly slow sorting method, and would be hopelessly inefficient in BASIC. However, in machine code the slowness is not a real drawback (1000 integers will be sorted in well under a minute) and there are compensating pedogogical features: the algorithm is easy to understand, and the program is short and generally economical in its use of storage locations.

LISTING 7.7

```
 10NUMBER=&70:FIRST=&71:SECOND=&73:TEMP=&75:BASE=&76:
   LOOPCOUNT=&78
 20DIM START 100
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA BASE
 60STA SECOND
 70LDA BASE+1
 80STA SECOND+1
 90LDA #0
100STA LOOPCOUNT
110.BEGIN
120LDY #0
130LDA SECOND+1
140STA FIRST+1
150LDA SECOND
160STA FIRST
170CLC
180ADC #4
```



```
190STA SECOND
200BCC NOCARRY
210INC SECOND+1
220.NOCARRY
230LDX #4
240SEC
250.LOOP1
260LDA (SECOND),Y
270SBC (FIRST),Y
280INY
290DEX
300BNE LOOP1
310BVC NOOVFLOW
320EOR #&80
330.NOOVFLOW
340EOR #0
350BPL OVER
360DEY
370.LOOP2
380LDA (FIRST),Y
390STA TEMP
400LDA (SECOND),Y
410STA (FIRST),Y
420LDA TEMP
430STA (SECOND),Y
440DEY
450BPL LOOP2
460.OVER
470INC LOOPCOUNT
480LDA LOOPCOUNT
490CMP NUMBER
500BNE BEGIN
510DEC NUMBER
520BNE START
530RTS:]NEXT
540CLS:INPUT"How many numbers",N:?NUMBER=N-1:DIM B 4
   *?NUMBER:!BASE=B
550FOR I%=0 TO N-1:!(B+4*I%)=RND:NEXTI%
560PRINT"Numbers assigned.  Sorting now":CALLSTART:
   PRINT"Done.  Checking now."
570FOR I%=0 TO N-2:IF  !(B+4*I%)>!(B+4+4*I%) THEN
   PRINT"ERROR AT "STR$(I%):END
580NEXTI% :PRINT"Checking O.K.":END
```

Listing 7.7 gives the program, the details of which are:

50-80 Put the base address (of where the integers are stored) initially in locations which will be used subsequently to point to the second integer in the bubble sort comparison.

90-100 Initialise the integer count. A location has to be used since the X and Y registers are used for other purposes.

120 Set the byte pointer within an integer (Y) to zero.

130-160 Put the old value of the pointer to the address of the second integer of the current pair into the pointer to the first integer of the pair.

170-210 Increment the second integer pointer by 4 so it points to the next integer.

230 Set the byte counter within an integer to four.

240-300 Subtract the first integer from the second integer, 'throwing away' the result except for the most significant byte.

310-320 If the overflow flag is set at the end of the subtraction, reverse the sign bit of the most significant byte (which is now in the accumulator).

340 This sets the negative flag to bit 7 of the contents of the accumulator. This is essential if no overflow has occurred, since 340 will be entered with the negative flag relating to DEX at 290. (ORA #0 or AND #&FF could have done equally well.)

350 If the second integer is less than the first, no need to swap them. (If we had subtracted the second from the first in 260 and 270, we would have had to swap equal integers. BMI OVER would not pick up equality, and there is no simple test for equality in the absence of the four bytes of the result.)

360-450 Swap the bytes of the integers one by one, starting with the most significant. Y already has the value 4 prior to line 360, so on entry to 380, Y = 3.

470-500 Continue until the current number of integers have been examined. Notice that 470 and 480 are more efficient than LDA LOOPCOUNT : CLC : ADC #1 : STA LOOPCOUNT.

510-520 Decrement the current number by one (the bubble principle) and continue until there are no integers left to bubble through.

You may be wondering why we cannot in 260-300 use CPY #4 and dispense with using X altogether. The reason is that CPY #4 affects the carry flag, and we need to leave the carry flag alone so it can register any borrowing which takes place. DEX does not affect the carry flag, so we are safe to use it. There is an alternative method which uses the stack, as we shall see in Chapter 9, but it is still not as efficient as the one we use here.

Notice the similarity between this program and the one in the last section. Although one deals with fixed length strings and the other with fixed length integers, the indirect indexed mode allows a common structure.

As usual we test the program using BASIC. Here we create a set of random integers using the RND function.

Exercise 7.3

Rewrite the program to deal with more than 256 integers.

## 7.6 SORTING A SERIES OF VARIABLE LENGTH STRINGS

We could try to use exactly the same structure as Listing 7.6 to sort variable length strings, but we will come upon a large difficulty when we try to swap them. Since they take up different quantities of memory, we will need to keep opening up and closing up sections of memory to accommodate the swap. One solution would be to store the strings in fixed spaces of 255 bytes (the maximum permissible) but this would be exceptionally wasteful of memory if we are not dealing with record structures where fixed lengths are practicable.

There is another way which is much more efficient: create a list of pointers and swap those instead. That is, store separately a list of the addresses of each of the strings, and instead of swapping the strings swap the addresses. Then, at the end of the sort, the first address will point to the string earliest in the alphabet, the second address to the string next in the alphabet, and so on. The strings themselves remain fixed in memory; it is the pointers which are moved about.

LISTING 7.8

```
 10NUMBER=&70:FIRST=&71:SECOND=&73:TEMP=&75:ADDRESS=&76:
   STORE=&78:STORE1=&7A:S TORE2=&7C
 20DIM START 9000: DIM BASE 512
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA BASE
 60STA ADDRESS
 70LDA BASE+1
 80STA ADDRESS+1
 90LDX NUMBER
100.LOOP1
110 LDY #&FF
120.LOOP2
130INY
140LDA (ADDRESS),Y
150CMP #&0D
160BNE LOOP2
170INY
180LDA STORE1
190CLC
200ADC #2
210STA STORE1
220BCC NOCARRY1
230INC STORE1+1
240.NOCARRY1
250CLC
260TYA
270LDY #0
```

```
280ADC ADDRESS
290STA (STORE1),Y
300STA ADDRESS
310BCC NOCARRY2
320INC ADDRESS+1
330.NOCARRY2
340LDA ADDRESS+1
350INY
360STA (STORE1),Y
370DEX
380BNE LOOP1
390.LOOP3
400LDA STORE
410STA STORE2
420LDA STORE+1
430STA STORE2+1
440LDX #0
450.LOOP4
460LDY #0
470LDA STORE2+1
480STA STORE1+1
490LDA STORE2
500STA STORE1
510CLC
520ADC #2
530STA STORE2
540BCC NOCARRY3
550INC STORE2+1
560.NOCARRY3
570LDA (STORE1),Y
580STA FIRST
590LDA (STORE2),Y
600STA SECOND
610INY
620LDA (STORE1),Y
630STA FIRST+1
640LDA (STORE2),Y
650STA SECOND+1
660LDY #0
670.LOOP5
680LDA (FIRST),Y
690CMP (SECOND),Y
700BCC NEWRECORD
710BNE SWAP
720INY
730BNE LOOP5
740.SWAP
750 LDY #1
760.LOOP6
770LDA (STORE1),Y
780STA TEMP
790LDA (STORE2),Y
800STA (STORE1),Y
810LDA TEMP
820STA (STORE2),Y
```

```
830DEY
840BPL LOOP6
850.NEWRECORD
860INX
870CPX NUMBER
880 BNE LOOP4
890DEC NUMBER
900BNE LOOP3
910RTS
920.TEXT:]NEXTI%
930!BASE=TEXT:!STORE=BASE:!STORE1=BASE
940INPUT'"How many records",N:?NUMBER=N-1
950PRINT'"Setting up strings now"
960 S=0:FOR I%=0 TO N-1:R=RND(40):FOR J%= 0 TO R-1:?
    (TEXT+S+J%)=RND(26)+64:NEXTJ%:?(TEXT+S+J%)=13
    :S=S+R+1:NEXTI%
970PRINT"Sorting now.":CALLSTART:PRINT"Checking."
980FOR I%=0 TO N-2:IF $(!(BASE+2*I%)MOD &10000)>$(!
   (BASE+2*I%+2)MOD &10000) THEN PRINT"ERROR AT "STR$(I%):END
990NEXT:PRINT"O.K.":END
```

The program is in Listing 7.8, and the details are as follows:

50-380 This section of the program sets up the list of pointers.

50-80 Put the base address of where the strings are to be stored into locations which will be used to point to the current string found. The address of the location BASE is set above the machine code program and the strings in line 20. This will be the starting point of the list of pointers. The first pointer (to the first string) is passed as data in line 930.

90 Initialise the string counter.

110-160 Scan the current string, starting with the first character, until the end of the string is found.

170 Set Y to the length of the string (including the carriage return). Recall that Y begins with zero, so the string length is Y + 1.

180 STORE1 will contain the address of the next vacant space in the list of pointers. On entry, it is set to contain the address of BASE at line 930.

190-230 Increment STORE1 by 2 so that it does now indeed point to the next vacant space in the pointer list.

250-280 Compute the low byte of the address of the next string in the list of strings by adding the length of the last string to its base address.

290 Store this low byte in the vacant space in the list of pointers, pointed to by STORE1 (set in lines 190-230). Y at this stage is zero.

300 Also store this low byte in the location which we are using to point to the current string.

310-320 Adjust the high byte of this pointer if carry has occurred.

340-360 Set Y to 1 and store the high byte in the vacant space next door to the low byte stored in 290.

370-380 Continue until all the strings have been accounted for.

We shall pause at this point, because this is a fairly complicated piece of coding. We have used indirect indexed addressing in two ways. Firstly, using (ADDRESS), Y, we have been able to examine, character by character, the contents of the current string being examined. Secondly, using (STORE1), Y, we have been able to access the next free location in the list of pointers. These are connected because we store the *contents* of ADDRESS in the memory location *pointed to* by the *contents* of STORE1. This idea will repay careful study, and you should not continue until you are confident you have fully grasped it. Symbolically, (ADDRESS) = ((STORE1), Y) when Y = 0 and (ADDRESS + 1) = ((STORE1), Y) when Y = 1.

The rest of the program does the sorting:

400-430 STORE contains the address of the first pointer in the list of pointers (i.e. it contains the value of BASE, *not* the contents of BASE). It is passed as data in 930. It is a permanent record of this address: STORE1, which did contain the address, has lost it in 50-380; and it will do so again in this part of the program.

440 Initialise the counter for the set of strings left to be sorted.

460 Initialise the string character index pointer.

470-500 Put the pointer to the location of the base address of the second string of the current pair being compared in the pointer to the first.

510-550 Increment the pointer to the second string address by 2 so it points to the next string address in the list of pointers.

570-650 Store the base addresses of the first and second strings of the current pair in locations on which indirect indexed addressing can be performed (in 680 and 690).

660-730 Compare the strings as in 220-300 of Listing 7.6.

770-840 If strings need swapping, swap their addresses. Remember STORE1 and STORE2 point to the *addresses* where the base addresses of the current strings are located, and not to the strings themselves.

860-910 Continue with the bubble sort until there are no strings left to bubble through.

Once again we check the program in BASIC. Here, random strings of random lengths up to 40 are used.

The ideas here are so important, and so tricky to grasp, that it is worthwhile finishing with a diagram to stress the relationship. Figure 7.2 considers a typical case. The current lower string of the pair begins at &2132. This address is contained in FIRST and FIRST + 1, whose contents were filled from the contents of &1682 and &1683. The first of these latter addresses is pointed to by the contents of STORE1 and STORE1 + 1. Illustrated is the flow of information required to access the sixth character (Y = 5) of the current lower string.

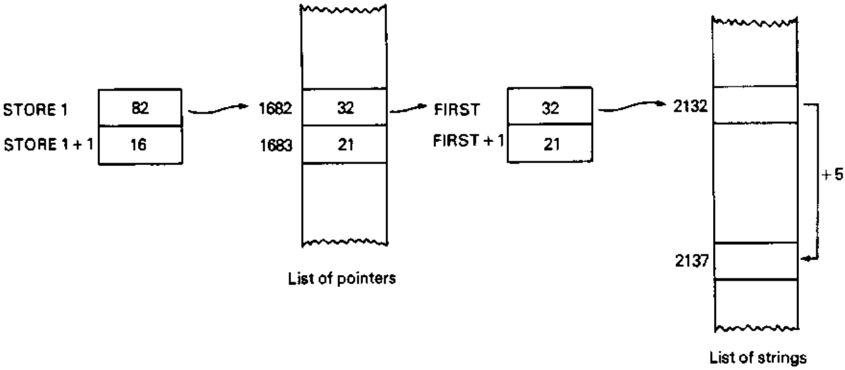


*Figure 7.2: The relationship between locations in listing 7.8.*

## Exercise 7.4

Rewrite the program to deal with more than 256 strings.

## 7.7 INDIRECT JUMPS AND JUMP TABLES

There is a non-indexed indirect addressing mode, available only with the JMP instruction. For example, the mnemonic JMP (&020E) will jump, not to &020E, but to the address *contained* in &020E and &020F (low byte first).

You are unlikely to want to use this in much of your programming. Its main use is in the structuring of operating system software, usually in conjunction with *jump tables*.

When you write JSR &FFEE for example (i.e. OSWRCH) the microprocessor goes through what seems a tortuous path. It goes to &FFEE and encounters JMP(&020E). So it picks up an address from

&020E and &020F and goes to the OSWRCH routine. Why this roundabout path? Surely it would be quicker to go straight to &E1BB, the address of the write character routine in OS.01?

When writing software, there are sometimes more important things than saving a few microseconds. There are two aspects to explain here: first the jump table, then the indirection. The BBC Computer has a series of operating system jumps stored from &FF00 onwards. These will either be straight jumps (i.e. 4C XX XX) or indirect jumps (i.e. 6C XX XX). In either case, there is a good reason for such a table  of jumps. The jumps in this table represent most of the important operating system calls.

Now, later on Acorn, the BBC micro's designers, may want to rewrite parts of the firmware (as software in ROM is often called). This may well involve changing some of the addresses for the important system calls. Since a jump table has been used, however, all references to the call need not change. All that needs to be changed is the reference address. So, for example, all calls to OSWRCH remain as &FFEE; all that needs to change is the contents of &020E and &020F (which is set up on power on or reset). Again, all references to JSR &FFA1 (which is not indirected) remain; all we need to change is the contents of &FFA2 and &FFA3. This makes operating system management a much easier task. And it ensures that all your programs which use operating system calls, will work with any change in the operating system.

Most of the important calls are indirected. One of the main reasons for this is to allow you to add pieces of code to the system software, whilst utilising the jump table idea. One example: suppose, when using OSRDCH, you always want to use the query prompt. Then you replace the contents of &0210 and &0211 (the indirection pointers for OSRDCH) by, say, &F8, &0D and write at &DF8 onwards:

```
LDA #ASC("?")
JSR OSWRCH
JMP &DFA5
```

Now all references to OSRDCH will indirect to &DF8.

The second reason for indirection is to allow you to intercept the operating system. As an example, suppose you wish to replace the prompt ">" by "?". If you type in and run the following program and then execute ?&20E = 0: ?&20F = &D, all occurrences of ">" will be replaced by "?".

```
10      FOR I% = 0 TO 2 STEP 2: P% = &D00
20      [OPT I%
30      CMP #ASC(">")
40      BNE OVER
50      LDA #ASC("?")
60    .OVER
70    JMP &E1BB: ]NEXT
```

To reset OSWRCH to its usual operation, execute ?&20E = &BB: ?&20F = &E1. (All absolute addresses are for OS 0.1)

Used with care, this facility can be of considerable use in application programs. FINDCODE, one of the utilities on the second tape available with this book, makes use of this to selectively list certain lines in a program.

Exercise 7.5

1. Small letters are always &20 more than capitals in their ASCII values (i.e. A is &41 and a is &61). Intercept OSWRCH to treat all small letters as capitals.

2. Use interception to give a query prompt to OSRDCH. (OSRDCH indirects through &0210 and its default value is &DFA5 in OS.01.)

# Chapter 8 Multiplication and division

## 8.1 A SIMPLE MULTIPLYING ALGORITHM FOR DECIMAL NUMBERS

Let us begin this chapter by reviewing the familiar algorithm for multiplying two base 10 numbers together.

Consider 1564 × 8401. We set out our working thus:

```
1564
    8401 ×
--------
    1564   (× 1)
   0000    (× 0)
  6256     (× 400)
12512      (× 8000)
--------
13139164
========
```

We take each digit of the multiplier 8401 in turn, beginning with the least significant (the right hand one), and multiply 1564 by it. As we move from right to left through the multiplier we move from right to left correspondingly through the columns. Thus each partial product is shifted one place to the left of the previous partial product. Then the partial products are added, column by column, to give the final result.

Each shift to the left is equivalent to multiplying by 10. Thus shifting twice and multiplying by 4 is equivalent to multiplying by 400; shifting three times and multiplying by 8 is equivalent to multiplying by 8000.

## 8.2 A CORRESPONDING ALGORITHM FOR BINARY NUMBERS

Consider now 1011 × 1101, both numbers being in binary. Our working is:

```
1011
    1101 ×
--------
    1011
   0000
  1011
 1011
--------
10001111
========
```

The algorithm is identical, but very much simpler to perform since we only ever multiply by 1 or 0. Here, each shift to the left is equivalent to multiplying by 2.

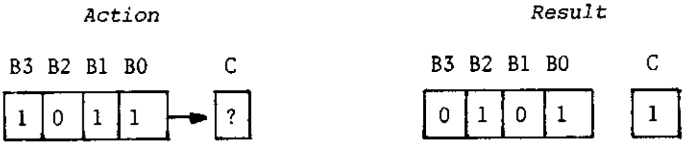
## 8.3 PROGRAMMING A 4-BIT MICROPROCESSOR TO PERFORM THE MULTIPLICATION ALGORITHM

Imagine that instead of an 8-bit microprocessor (i.e. one with an 8-bit data bus and 8-bit registers) we have a 4-bit microprocessor (with 4-bit data bus and registers). How can we program it to perform the multiplication of 4-bit numbers? Once we have solved this problem for a 4-bit processor, it will be an easy matter to extend it to an 8-bit one.

Let us discover first what programming commands we have at our disposal. There are four:

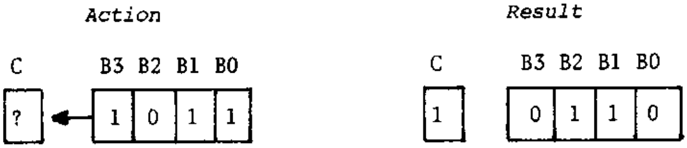
(i) SHIFT RIGHT

This shuffles all the bits along one place to the right. The space created at the most significant end is filled with a zero. The bit which falls out at the right is put into the carry flag. Consider what happens to 1011. It becomes 0101 and 1 goes into the carry. Thus:



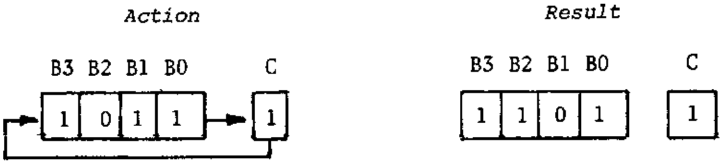
(ii) SHIFT LEFT

Identical to shift right except all the bits are shuffled one place to the left, the least significant bit being filled with a zero, the most significant falling into the carry flag. Thus on 1011 we have:

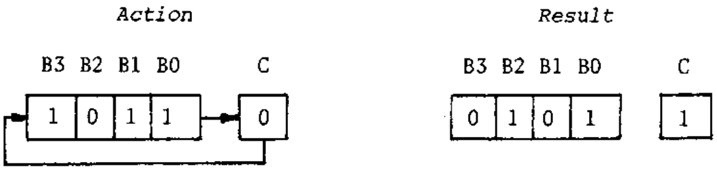


(iii) ROTATE RIGHT

This is just like shift right except that the space created at the most significant end is filled by whatever was in the carry. Thus, again considering 1011, if the carry contains a 1 we have:

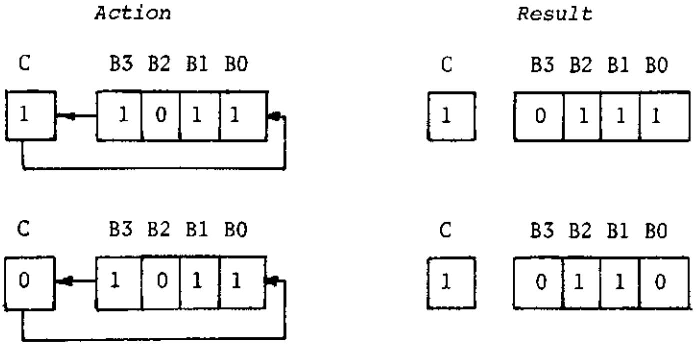


If the carry contains a zero we have:



(iv) ROTATE LEFT

Identical to rotate right except the movement is leftwards. The corresponding diagrams are:



These four operations can act on any memory location, zero page or absolute. In addition, they can also act directly on the accumulator. This is something new, and constitutes an additional form of addressing, called, not surprisingly, *accumulator addressing*. In this case the operand is simply the accumulator itself.

Now, let us use these instructions to model the algorithm as closely as possible.

We will need two four bit locations for our result: RESH for the top four bits and RESL for the bottom four bits (remember that at the moment we are assuming that memory locations hold only four bits). We will need a temporary location, TEMP; you will see why in a moment. Finally we need a place to store the multiplier, MULTER, and a place to store the number multiplied by this, MULTED. In our example above MULTER is 1101 and MULTED is 1011.

We begin by setting RESL, RESH and TEMP equal to zero. We now look at the least significant bit of MULTER. If it is one we add MULTED to RESL. We then shift MULTED one place to the left, putting the left next bit which drops out into TEMP. Look at the next bit of MULTER. If one, add MULTED to RESL and TEMP to RESH (together with any carry). Continue in this way through all four bits of MULTER.

The flow chart of the process is in Fig.8.1 (we assume MULTER and MULTED already have their values assigned). Recall that (MULTER$_I$) refers to bit I of MULTER.

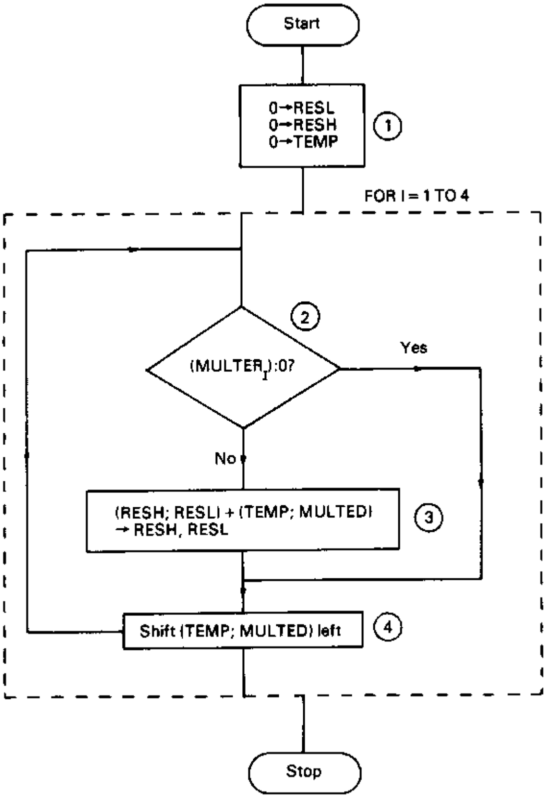


*Figure 8.1*

| MULTER | RESH | RESL | TEMP | MULTED | |
|---|---|---|---|---|---|
| 1101 | 0000 | 0000 | 0000 | 1011 | Initialisation |
| 0110 [c = 1] | 0000 | 1011 | 0001 | 0110 | Loop 1 |
| 0011 [c = 0] | 0000 | 1011 | 0010 | 1100 | Loop 2 |
| 0001 [c = 1] | 0011 | 0111 | 0101 | 1000 | Loop 3 |
| 0000 [c = 1] | 1000 | 1111 | 1011 | 0000 | Loop 4 – End |

*Table 8.1*

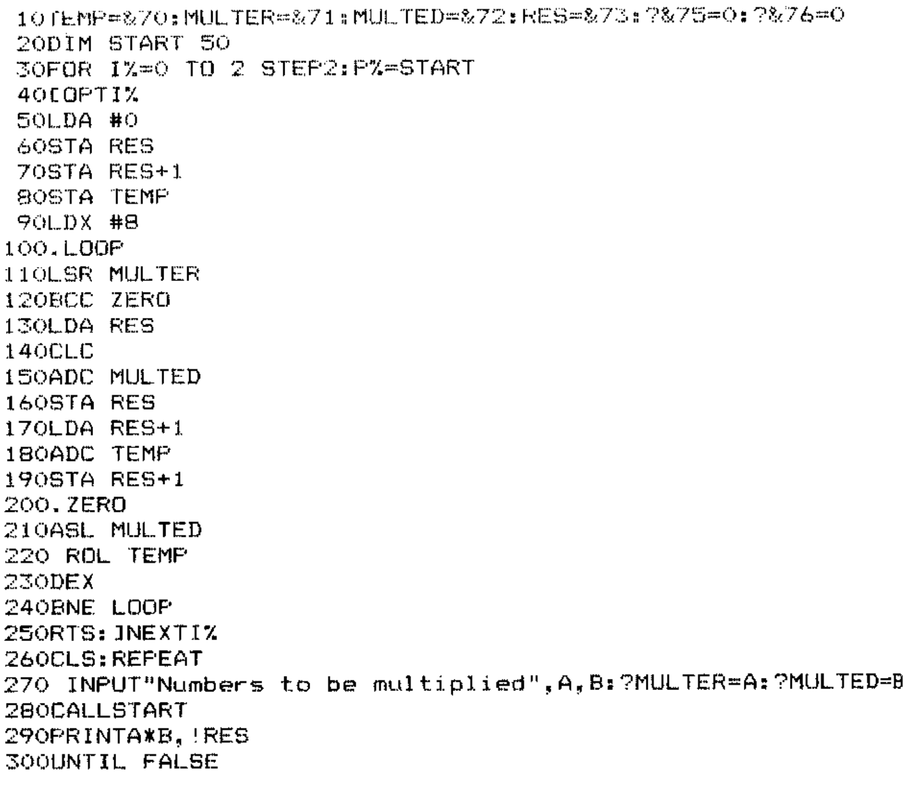
The only problem now concerns how we can test the successive bits of MULTER. This is most easily done by shifting right each time; the next bit of MULTER will fall into the carry flag and we can test this flag to see whether it is zero.

Since the ideas here are so important, let us follow through the flow chart for 1011 × 1101. The analysis is in Table 8.1.

## 8.4 A PROGRAM TO MODEL THE MULTIPLICATION ALGORITHM

Now we have understood the process using the simpler 4-bit microprocessor we can now write the program for our 8-bit microprocessor. The flow chart will be identical to Figure 8.1 except the loop is from 1 to 8.

LISTING 8.1



```
10TEMP=&70:MULTER=&71:MULTED=&72:RES=&73:?&75=0:?&76=0
20DIM START 50
30FOR I%=0 TO 2 STEP2:P%=START
40[OPTI%
50LDA #0
60STA RES
70STA RES+1
80STA TEMP
90LDX #8
100.LOOP
110LSR MULTER
120BCC ZERO
130LDA RES
140CLC
150ADC MULTED
160STA RES
170LDA RES+1
180ADC TEMP
190STA RES+1
200.ZERO
210ASL MULTED
220 ROL TEMP
230DEX
240BNE LOOP
250RTS:]NEXTI%
260CLS:REPEAT
270 INPUT"Numbers to be multiplied",A,B:?MULTER=A:?MULTED=B
280CALLSTART
290PRINTA*B,!RES
300UNTIL FALSE
```

The program is in Listing 8.1. As usual we have used RES and RES + 1 for a two byte label (instead of RESL and RESH). In this listing three of the mnemonics for the shifts and rotates are introduced. LSR is shift right; ASL is shift left; ROL is rotate left; and ROR, the one missing, is rotate right. Appendix 1 gives the symbolic diagrams and addressing modes. Note that LSR denotes *logical* shift right and ASL *arithmetic* shift left. These first words are of no consequence and are best ignored. So remember LSR as shift right and ASL as shift left; and to distinguish, think of shifting left as multiplying by 2 and so arithmetic.

The important thing to notice in Listing 8.1 is the method of shifting TEMP; MULTED left. Shift MULTED left (ASL); then shift TEMP left rotating in the carry flag (ROL). Thus we use the carry flag as an intermediary in shifting the MSB of MULTED into the LSB of TEMP. This technique is used a great deal and should be thoroughly understood and committed to memory.

## 8.5 A MORE EFFICIENT ALGORITHM FOR MULTIPLICATION

Our program in 8.4 models very accurately the algorithm in 8.2. But are we sure that algorithm makes best use of the microprocessor's facilities? Let us return again to the 4-bit machine of 8.3.

Consider the following method of performing 1011 × 1101:

(1) Begin with 1011 0000 in RESH; RESL, since LSB of 1101 is 1. Shift right to 0101 1000.

(2) Since LSB of 0110 is 0 there is nothing to add on. Shift 0101 1000 right to 0010 1100.

(3) Since LSB of 0011 is 1 add 1011 0000 to 0010 1100, giving 1101 1100. Shift right to 0110 1110.

(4) Since LSB of 0001 is 1 add 1011 0000 to 0110 1110 giving 0001 1110 and 1 in carry. Rotate right to 1000 1111.

The great advantage here is that we do all our shifting in RESH and RESL and thus have no need for TEMP. As we move through the four stages of this process, the contents of RESH; RESL make the following changes:

(1) 0101 1000 (2) 0010 1100

$$\text{(3)}\quad \begin{array}{rl} & 0010\ 1100 \\ & \underline{1011\ 0000} + \\ & 1101\ 1100 \\ \to & 0110\ 1110 \end{array} \qquad\qquad \text{(4)}\quad \begin{array}{rl} & 0110\ 1110 \\ & \underline{1011\ 0000} + \\ \boxed{1} & 0001\ 1110 \\ \to & 1000\ 1111 \end{array}$$

The logic here is that we add on each partial product before shifting the partial result. In this way each partial product is shifted once less than the partial product immediately preceding it, and thus ends up in the correct column. The first partial product we take is the one which will end up furthest to the right; the next one is the one which will end up one less column to the right; and so on.

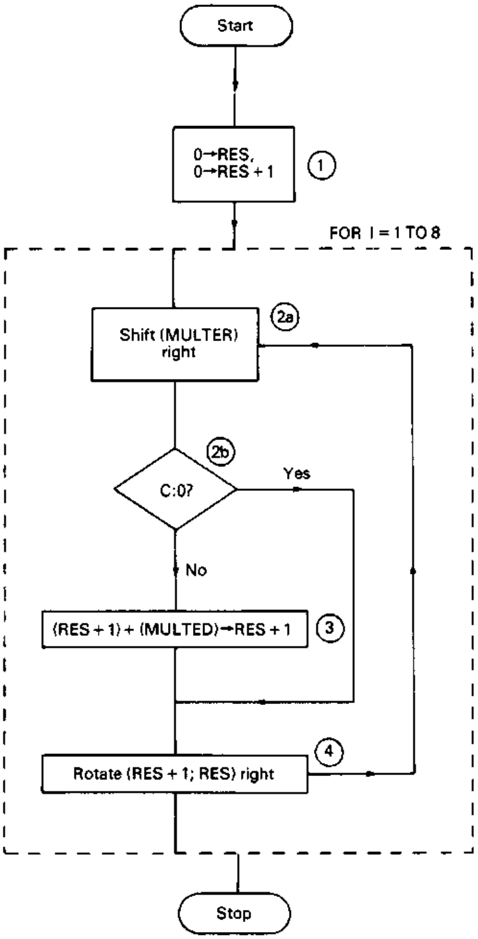


*Figure 8.2*

LISTING 8.2

```
 10MULTER=&70:MULTED=&71:RES=&72:?&74=0:?&75=0
 20DIM START 50
 30FOR I%=0 TO 2 STEP2:P%=START
 40[OPTI%
 50LDA #0
 60STA RES
 70STA RES+1
 80LDX #8
 90.LOOP
100LSR MULTER
110BCC ZERO
120LDA RES+1
130CLC
140ADC MULTED
150STA RES+1
160.ZERO
170ROR RES+1
180ROR RES
190DEX
200BNE LOOP
210RTS:]NEXTI%
220CLS:REPEAT
230 INPUT"Numbers to be multiplied",A,B:?MULTER=A:?MULTED=B
240CALLSTART
250PRINTA*B,!RES
260UNTIL FALSE
```

Figure 8.2 gives the flowchart for an 8-bit processor, and Listing 8.2 gives the program. Note the use of ROR RES+1 instead of LSR. This ensures that any 'ninth bit' is rotated in (a situation analogous to stage (4) in the 4-bit processor algorithm above).

## 8.6 MORE EFFICIENCY STILL: ACCUMULATOR ADDRESSING

We met the idea of accumulator addressing in 8.3. If you inspect Listing 8.2 you will notice that after initialisation the accumulator is used only to load and store RES+1. We can therefore do without RES+1 until the end and perform the shifting on the accumulator instead. This has the bonus of being quicker too (2 cycles instead of 3).

LISTING 8.3

```
10MULTER=&70:MULTED=&71:RES=&72:?&74=0:?&75=0
20DIM START 50
30FOR I%=0 TO 2 STEP2:P%=START
40[OPTI%
50LDA #0
60STA RES
70LDX #8
80.LOOP
90LSR MULTER
100BCC ZERO
110CLC
120ADC MULTED
130.ZERO
140ROR A
150ROR RES
160DEX
170BNE LOOP
180STA RES+1
190RTS:]NEXTI%
200CLS:REPEAT
210 INPUT"Numbers to be multiplied",A,B:?MULTER=A:?MULTED=B
220CALLSTART
230PRINTA*B,!RES
240UNTIL FALSE
```

The new listing is in Listing 8.3. Note that the mnemonic label for accumulator addressing is A - hence A cannot be used as a location name. As a final point, if we did not want to store the result, only output it to the screen (as we will do in the next section and in Chapter 9) then we could dispense with line 180 also.

## Exercise 8.1

1. Adapt Figure 8.2 to deal with the multiplication of two 16-bit numbers. You will need four locations for the result: RES+3, RES+2, RES+1, RES. Also two bytes for the multiplier, MULTER+1 and MULTER, and two for the multiplied number, MULTED+1 and MULTED.

Write the corresponding program. Can you make it more efficient by using the accumulator as a storage location for part of the result?

2. Adapt Listing 8.3 to multiply two signed 8-bit numbers. You will need to check if either number is negative. If it is, you will need to form the two's complement. You will need to determine the sign of the final result also. This can be combined with the earlier checks by initially setting X to zero. Increment X if the first number is negative, decrement X if the second is negative. Now if X is zero the result is positive; otherwise it is negative (set Y to 1 in this case).

Now Listing 8.3 can be used, and the sign of the result adjusted accordingly.

3. Write a program to compute 250 * Y + X (see 5.4b).

## 8.7 AN INTERLUDE: OUTPUTTING NUMBERS USING BINARY CODED DECIMAL

In Chapter 6 we saw how to output characters to the screen, but we did not consider how we could output a number stored in a byte. For example, how can we output the byte &A3 to the screen as the decimal number 164? We will consider a quite general way of doing this in the next chapter, but here we will combine some of the new instructions in the last few sections with a new way of storing data: binary coded decimal (or BCD).

Any byte consists of two sets of four bits, the four least significant and the four most significant. With their customary humour, computer scientists refer to these half-bytes as *nybbles*! We can conceive of the bottom nybble as representing a decimal digit, and the top nybble as another decimal digit. No problems will arise as long as we restrict the range of each nybble from binary 0000 to binary 1001 (i.e. from 0 to 9). Thus we could understand 00110100 as decimal 34 in this approach. To put this another way, as long as each hex digit is restricted to between 0 and 9, we can interpret a hex byte as a two digit decimal number. Then, a byte can now represent any of the decimal numbers from 0 to 99. This way of coding decimal numbers is called, for obvious reasons, *binary coded decimal* (BCD).

Now when we want to perform addition or subtraction using BCD numbers, we want the computer to behave in a special way. As an example, let us add 78 to 34 in BCD. We have:

| | |
|---|---|
| 34 | 0011 0100 |
| 78 | 0111 1000 |

Without any special changes, the computer will produce 1010 1100 which is obviously rubbish in BCD. What we would like the computer to do is to generate a carry after it adds the lower two nybbles, correcting the result to BCD: thus,

$$\begin{array}{lr} & 0100 \\ & \underline{1000} \\ \text{Carry 1} & \underline{0010} \end{array}$$

We want now to be able to add this carry with the addition of the higher two nybbles, and again generate a new carry correcting the result to BCD: thus

$$\begin{array}{lr} & 0011 \\ & 0111 \\ & \underline{1} \\ \text{Carry 1} & \underline{0001} \end{array}$$

The final result will be $\boxed{1}$ 12 in BCD (with the hundreds digit in the carry), which is correct.

There is a simple way to get the computer to behave like this: we just set a flag in the processor status register called the D flag. The instruction is SED.

Whilst this flag remains set all occurrences of ADC and SBC are treated as BCD. Then an internal carry from the lower to the upper nybble will automatically occur as the lower nybble passes through 9, and it is automatically set back to 0. Similarly a carry to the carry flag is generated automatically as the upper nybble passes through 9, and this nybble also is set back to 0. Hence if we add 1 to itself 100 times we will obtain 00 with a carry of 1.

It is this idea that we exploit to convert our byte to a set of ASCII digits which we can output to the screen. The idea is to decrement the byte to be converted (using DEX, with the byte in X) and increment the accumulator by 1 in BCD (using ADC #1 with the D flag set). The highest digit can then be straightforwardly output (&30 + digit). The lower digits have to be separated: we use LSR A four times to shift the top nybble of the accumulator into the bottom nybble position, at the same time setting the top nybble to zero. Then again output in ASCII. Finally, we reclaim the accumulator, perform AND #&0F to set the top nybble to zero whilst leaving the bottom nybble unchanged; and then output in ASCII.

LISTING 8.4

```
 10HIDIGIT=&70:NUMBER=&71:OSWRCH=&FFEE:OSNEWL=&FFE7
 20DIM START 100
 30FOR I%= 0 TO 2 STEP2: P%=START
 40[OPTI%
 50LDA #&30
 60STA HIDIGIT
 70LDA #0
 80LDX NUMBER
 90BEQ ZERO
100SED
110CLC
120.BACK
130ADC #1
140BCC NOCARRY
150INC HIDIGIT
160CLC
170.NOCARRY
180DEX
190BNE BACK
200CLD
210.ZERO
220TAY
230LDA HIDIGIT
240JSR OSWRCH
250TYA
260LSR A
270LSR A
280LSR A
290LSR A
300CLC
310ADC #&30
320JSR OSWRCH
330TYA
340AND #&0F
350ADC #&30
360JSR OSWRCH
370JSR OSNEWL
380RTS:]NEXTI%
390CLS:REPEAT
400INPUT"Number to be output",?NUMBER
410CALLSTART
420UNTIL FALSE
```

The program is in Listing 8.4, and the details are:

50-60 Store the ASCII code for zero in the location for the highest digit.

70-90 Initialise the accumulator and the X register. If the byte to be converted is zero, skip the BCD conversion.

100-110 Set the relevant flags for the conversion.

130-190 Add 1 in BCD to the accumulator and at the same time decrement the byte in X. If the accumulator passes from 99 to 00 increment the high digit. Continue until the byte reaches zero.

200 Clear the D flag. It is most important to do this as soon as you are finished with BCD arithmetic, and also whenever you want to use a system subroutine like OSWRCH.

220-250 Save a copy of the accumulator in Y, output the highest digit to the screen, and reclaim the accumulator (the copy is still in Y).

260-320 Set the top nybble into the bottom nybble position and set the top nybble to zero. Add the ASCII code and output to the screen.

330-380 Get the accumulator back (stored in 220), set the top nybble to zero, add the ASCII code, output the digit to the screen, output a new line and return.

This routine will always display three digits: 000 to 255. It can be adapted to output the contents of two or more bytes, but it is not really worth it since the next chapter contains a much more efficient routine. However, it is very straightforward to adapt it to count from 000 to 999 and this is left as an exercise.

You may be wondering why we do not use BCD to multiply, since we are so accustomed to base ten work. The problem is that BCD is very wasteful of space, and it is slow because many more byte movements are involved e.g. 193 × 253 will involve 3 instead of two bytes, and each multiplication by ten involves four shifts which multiplies by 16 in binary.

### Exercise 8.2

What amendments to Listing 8.4 are required to allow it to output numbers from 000 to 999?

## 8.8 A SECOND INTERLUDE: A PSEUDO-RANDOM NUMBER GENERATOR

Here is a relatively unknown, but very simple and very fast way to produce your own pseudo-random numbers.

We begin with a 32-bit shift register, which we seed initially when assembling the program with a 32-bit pseudo-random number from BASIC's generator (i.e. we use RND).

Every time the program is called, this 32-bit register is 'specially' rotated left 8 bits, and the number remaining in the highest 8-bits is the random number. The 'special' aspect of the shifting lies in this: if the most significant bit of the register is 0 we simply rotate the register left one bit; if the most significant bit of the register is 1, we first exclusive-or the bottom three bytes with a suitable constant before rotating left one bit. The constant we will use here is, highest byte first, &76, &B5, &53, but others are possible.

At the end of 8 rotations we have our random number, which we put into the accumulator. The program is given in Listing 8.5. Lines 280 to 370 perform a chi-square test on some test data. When I ran this with 500 sets of 256, the null hypothesis of randomness could not be rejected even at a significance level of 25%. This is strong evidence in favour of randomness. Permutation tests also show that there is no clustering. Moreover, it can be shown that repetition occurs only after $2^{29}$ bytes (i.e. after 536,870,912 bytes). This is therefore a very good pseudo random generator,

LISTING 8.5

```
 10CONSTANT =&70:SHIFTREG=&73
 20DIM START 50
 30FOR I%= 0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDY #8
 60.BEGIN
 70CLC
 80LDA SHIFTREG+3
 90BPL ZEROBIT
100LDX #2
110.LOOP
120LDA SHIFTREG,X
130EOR CONSTANT,X
140STA SHIFTREG,X
150DEX
160BPL LOOP
170SEC
180.ZEROBIT
190ROL SHIFTREG
200ROL SHIFTREG+1
210ROL SHIFTREG+2
220ROL SHIFTREG+3
230DEY
240BNE BEGIN
250LDA SHIFTREG+3
260RTS:]NEXTI%
270?&70=&53:?&71=&B5:?&72=&76:!&73=RND:
   REM ***Initialisation***
280VDU12:INPUT"How many sets of 256 for the test",T%
290DIM N%(255):VDU12:FOR I%=1 TO T%
300FOR J%=1 TO 256:A%=USRSTART:A%=?&404
310N%(A%)=N%(A%)+1:NEXT
320PRINTTAB(0,1)I%:NEXT
330S%=0:FORI%=0 TO 255:S%=S%+N%(I%)^2:NEXT
340PRINT"Chi-squared gives ";S%/T%-256*T%
350PRINT"This compares with :"'FNCHI(1.64)" at 5%
   "'FNCHI(1.28)" at 10%"'FNCHI( 0.84)" at 20% and
   "'FNCHI(.675)" at 25%"
360END
370DEF FNCHI(X) = 0.5*(X + 22.56)^2
```

especially given its simplicity and speed. By calling it four times, you can, if you wish, generate 32 bit signed integers equivalent to RND, with a repetition cycle of 134,217,728.

The program requires seven memory locations, all of which are initialised on assembly in line 270. Thereafter, no further initialisation is required. The details are:

50 Initialise bit count.

70 If most significant bit is zero, we require zero in the carry flag when rotating into the lowest byte of the shift-register at line 190.

80-90 If msb is zero, no need to exclusive-or.

100-160 Exclusive-or bottom three bytes of shift-register with constant.

170 Set carry to one for rotation in line 190.

190-220 Rotate shift-register one bit to the left.

230-260 Continue for 8 bits, load random number into the accumulator and return.

Notice that we must initially load the carry flag with zero or one to perform the 32-bit rotation, since the rotation into the lowest bit will need to contain the contents of the highest bit.

## 8.9 A THIRD INTERLUDE: COPYING THE HIGH-RESOLUTION SCREEN TO A PRINTER

As a third example involving our new instructions, we will consider how we can arrange to output the high-resolution screen to a printer. Specifically we shall focus on MODE 4 and MODE 0, since these are the two-colour modes where the maximum resolution is possible. The program is written for an EPSON MX80 type II printer, but even if you lack one of these, or even indeed if you have only a model A with no printer interface, the ideas here will still be of use to you. From the programming point of view the techniques involved are of interest, and the information on how the BBC computer organises its high resolution graphics should be valuable too.

In MODE 4 it is helpful to think of the screen as 32 rows, each row containing 320 bytes consisting of 8 rows and 40 columns. In MODE 0, 32 rows again, but this time 640 bytes consisting of 8 rows and 80 columns.

As long as no scrolling has occurred since the last clear screen or mode change (this will typically be the case when we are plotting on the high-resolution screen) then the first byte of screen memory will be the first byte of the first row and first column of the screen. What happens if scrolling occurs (which will typically only happen in MODEs 0 and 4 if we are using user-defined characters textually) will be covered in Chapter 9.

Thereafter, the memory is allocated as follows: we remain in row one, scan the first column of 8 bytes, then the second and so on to the eightieth (or fortieth) column. Then onto row two, and repeat the scan; then row three, etc. until we end at row 32. Figure 8.3 should make this clear. With no scrolling, the final byte at the bottom right-hand corner is &7FFF (or &3FFF in a Model A). In multi-colour modes, part of each byte contains the logical colour information but we are not concerned with this here.

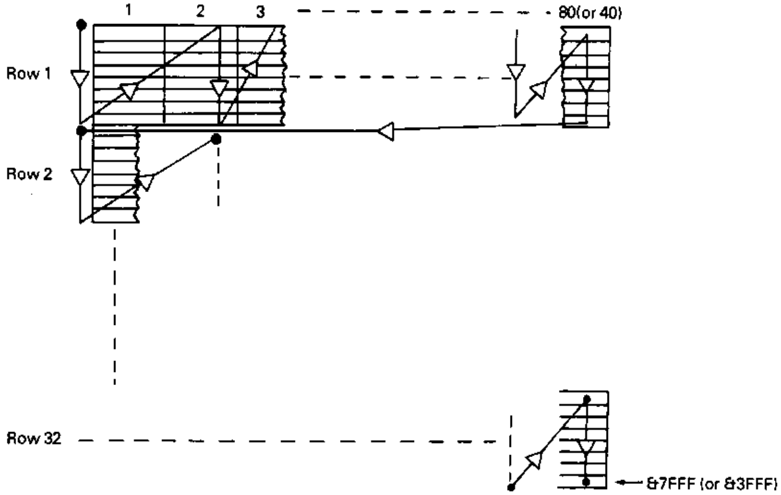


*Figure 8.3* The organisation of screen memory on the high-resolution screen in MODE 0 (or 4) when no scrolling has occurred since the last clear screen.

Now in order to understand how we can copy the screen to the printer, we must understand how the Epson printer prints high resolution graphics (other high-resolution printers operate similarly).

When the right control codes are passed to it (more on this in a moment) the Epson will treat each byte of data passed to it as specifying which of its 8 needles will fire (the ninth, bottom needle, will never fire in high-resolution mode). The top needle corresponds to the most significant bit of the byte of data, the eighth needle to the least significant. Thus &B3, which is 10110011 in binary, will fire the top needle, the third and fourth needles down, and the seventh and eighth needles down. The pattern produced is shown in Figure 8.4: notice it is a vertical pattern.

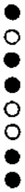


*Figure 8.4: The pattern produced on an EPSON printer with the data &B3.*
*● indicates a dot on the paper (i.e. needle fires)*
*o indicates a space on the paper (i.e. needle does not fire)*

Now, in order to get the correct orientation on the paper it is necessary to output each of the 32 rows, column by column. That is, the first 8 × 8 bits must be output not in rows but in columns: Figure 8.5 should make this clear. This is where our new instructions come in: we need to extract the column information from the 8 bytes which contain the row information.

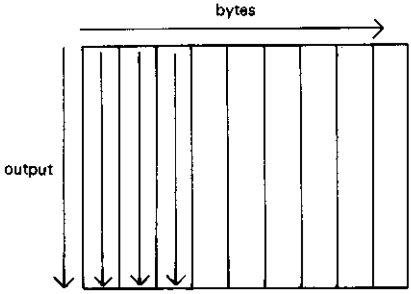


*Figure 8.5: An 8 × 8 set of bits at the intersection of any row and column. The bytes relate to the rows of this grid, but the output must relate to the columns so that it matches the set-up in figure 8.4.*

LISTING 8.6

```
 10COLUMNS=&70:ROWS=&71:COLCOPY=&72:LIMIT=&73:
   BEGINCONTROL=&74:LOCATION=&75:STORE=&77:OSWRCH=&FFEE:
   OSBYTE=&FFF4
 20FORI%=0 TO 2 STEP 2:P%=&D00:RESTORE
 30[OPTI%
 40LDA #3
 50STA LIMIT
 60LDX #0
 70JSR CONTROL
 80LDX #4
 90LDA #&85
100JSR OSBYTE
110STY MEMLOC+1
120LDA #&84
130JSR OSBYTE
140STX LOCATION
150STY LOCATION+1
160.MEMLOC
170CPY #0 Dummy operand
180BNE ZEROMODE
190LDA #7
200STA LIMIT
```

```
210LDA #3
220STA BEGINCONTROL
230LDA #40
240BNE FOURMODE
250.ZEROMODE
260LDA #11
270STA LIMIT
280LDA #7
290STA BEGINCONTROL
300LDA #80
310.FOURMODE
320STA COLUMNS
330LDA #32
340STA ROWS
350.BEGIN
360LDA COLUMNS
370STA COLCOPY
380LDX BEGINCONTROL
390JSR CONTROL
400.LOOP1
410LDY #7
420.LOOP2
430LDA (LOCATION),Y
440STA STORE,Y
450DEY
460BPL LOOP2
470LDY #8
480.LOOP3
490LDX #7
500LDA #1
510JSR OSWRCH
520.LOOP4
530ASL STORE,X
540ROR A
550DEX
560BPL LOOP4
570JSR OSWRCH
580DEY
590BNE LOOP3
600LDA LOCATION
610CLC
620ADC #8
630STA LOCATION
640BCC NOCARRY
650INC LOCATION+1
660.NOCARRY
670DEC COLCOPY
680BNE LOOP1
690LDA #1
700JSR OSWRCH
710LDA #&0D
720JSR OSWRCH
730DEC ROWS
740BNE BEGIN
750LDA #13
```

```
760STA LIMIT
770LDX #11
780JSR CONTROL
790RTS
800.CONTROL
810LDA #1
820JSR OSWRCH
830LDA TABLE,X
840JSR OSWRCH
850INX
860CPX LIMIT
870BNE CONTROL
880RTS
890.TABLE:]NEXTI%
900FOR I%=1 TO 13
910READ ?P%
920P%=P%+1:NEXTI%
930DATA27,65,8,27,75,64,1,27,76,128,2,27,80
```

The program is shown in Listing 8.6, and the details are:

40-70 Output the first three bytes in the data statement (at 930) to the printer. This is ESCA 8 and it ensures that there will be no gaps between the lines output (i.e. line feed is exactly 8 dots in depth).

80-110 Using OSBYTE with &85 in the accumulator returns the first memory location in screen memory with the mode number in the X register (here Mode 4). The low byte goes into X and the high byte into Y. Here we store Y (at 170) for future comparison.

120-150 OSBYTE with &84 in the accumulator returns the first memory location in screen memory for the current actual mode (low in X, high in Y). We store this address permanently into LOCATION.

170-180 Since we assume the program will be used in either MODE 0 or MODE 4, if the comparison in 170 is not equal (i.e. high byte of current screen memory is not the same as it would be in MODE 4), we assume MODE 0 operates.

190-220 If MODE 4, output ESCK 64 1 to the printer. ESCK is the normal density high resolution mode, which gives a maximum of 480 bits (i.e. 60 columns of 8 bits/column) per line. This is adequate for MODE 4. 64 1 is 64 + 1 × 256 i.e. 320, and this tells the printer we will output 320 bits (i.e. 40 columns of 8 bits/column) per line.

230-240 Put the column count in the accumulator and always branch to 320.

260-290 In MODE 0, output ESCL 128 2 to the printer, which is double density high resolution (i.e. maximum of 960 bits or 120 columns of 1 byte/column) per line. We inform the printer that we will output 128 + 2 × 256 = 640 bits (i.e. 80 columns) per line.

300 Put column count for MODE 0 in the accumulator.

320-340 Store the relevant column count in COLUMNS; ROWS is always 32 for either mode.

360-370 COLUMNS will be required later, so store a copy in COLCOPY, which we can alter.

380-390 BEGINCONTROL contains either 3 in MODE 4 (from 220) or 7 in MODE 0 (from 290). LIMIT is fixed from 200 or 270. Thus we output the same control information per line i.e. ESCK 64 1 or ESCL 128 2.

410-460 Put the 8 bytes of the current column into STORE. Notice in 440 we forego the one byte saving available in zero page with STORE, X. However, there is only a small time penalty and it is overall much quicker to do this than to enlist X as well (we must use Y for line 430).

470 Initialise the byte counter (Y) for output to printer.

490 Initialise the bit counter (X) for each byte output to printer.

500-510 The next output will be to the printer only.

530-570 Taking each byte in turn in STORE (moving backwards) shift out a byte to the left and rotate it right into the accumulator. After the 8 bytes in store have been shifted once in this way the accumulator will contain 8 bits, the most significant relating to the top byte (i.e. STORE, 0) and the least significant relating to the bottom byte (i.e. STORE, 7). This accords with Figures 8.4 and 8.5.

580-590 Continue to shift until 8 bytes are output (i.e. the 8 columns of Figure 8.5).

600-650 Increment the screen memory location to the next column.

670-680 Continue for the 80 (or 40) columns.

690-720 Output to the printer a CR to mark the end of the line. (This assumes your printer is set up so that CR generates an automatic LF.)

730-740 Continue until all 32 rows are covered by returning to 360.

750-790 Output some control characters which reset the line feed to its usual depth, and return.

810-880 Outputs selected bytes from a data statement to the printer only.

Notice that this program assembles into &D00. This is useful in that it can now be combined with any BASIC program beginning in &E00 by using *SAVE between the memory limits 0D00 and TOP. Now *LOAD will pick up the machine code and the BASIC program, and OLO will initialise the BASIC pointers. An example of this is contained on TAPE 1 available with this book. This is a double function graph plotting program, and the screen can be copied to a printer by simply pressing P which activates a CALL &D00 in the program.

## 8.10 DIVISION

Let us examine how we would divide 170 by 28 in binary. This is 10101010 ÷ 00011100. We shall use the familiar algorithm for long division, written a little more fully than usual.

```
DIVISOR  DIVIDEND       QUOTIENT
    11100)10101010     (0 1 1 0     (a) Dividend < divisor so
          11100000  (a)                 put zero in quotient
                                        and do not subtract
DIVIDEND  10101010                      divisor from dividend.
DIVISOR   01110000  (b)

DIVIDEND  00111010                  (b) Dividend > divisor so
DIVISOR   00111000  (b)                 put one in quotient and
                                        subtract divisor from
DIVIDEND  00000010                      dividend.
DIVISOR   00011100  (a)

DIVIDEND        10 remainder
```

The general idea here is to start at the most significant end of the dividend (the number into which we are dividing) and compare the 5 most significant bits with the five in the divisor. This is equivalent to attempting to subtract from the dividend the number 11100000 and seeing whether the result is positive. If it is, we put a one in the most signficant bit of the quotient (the result); if not we put zero. If we put a one, we replace the quotient by the result of the subtraction; if we put a zero, we leave the quotient as it is.

We now shift the divisor one place right to give 01110000, and repeat the above process. We continue until we have shifted across to the original divisor, 00011100, and after we have performed the compare and subtraction process on that we stop. In order to know how many right shifts we must do before stopping, we start by shifting 11100 left until a one appears in bit 7. We store a count of the number of left shifts necessary and reverse this count when shifting right.

The flowchart for the process is in Figure 8.6, where we use the X-register to count the shifts required. Notice how we ensure that the quotient has its digits in the correct place: we shift left for every shift right of the divisor, so that the digits automatically line up. The program is in Listing 8.7. Notice that we have reversed the shift of (DVIS) and the incrementing of X in

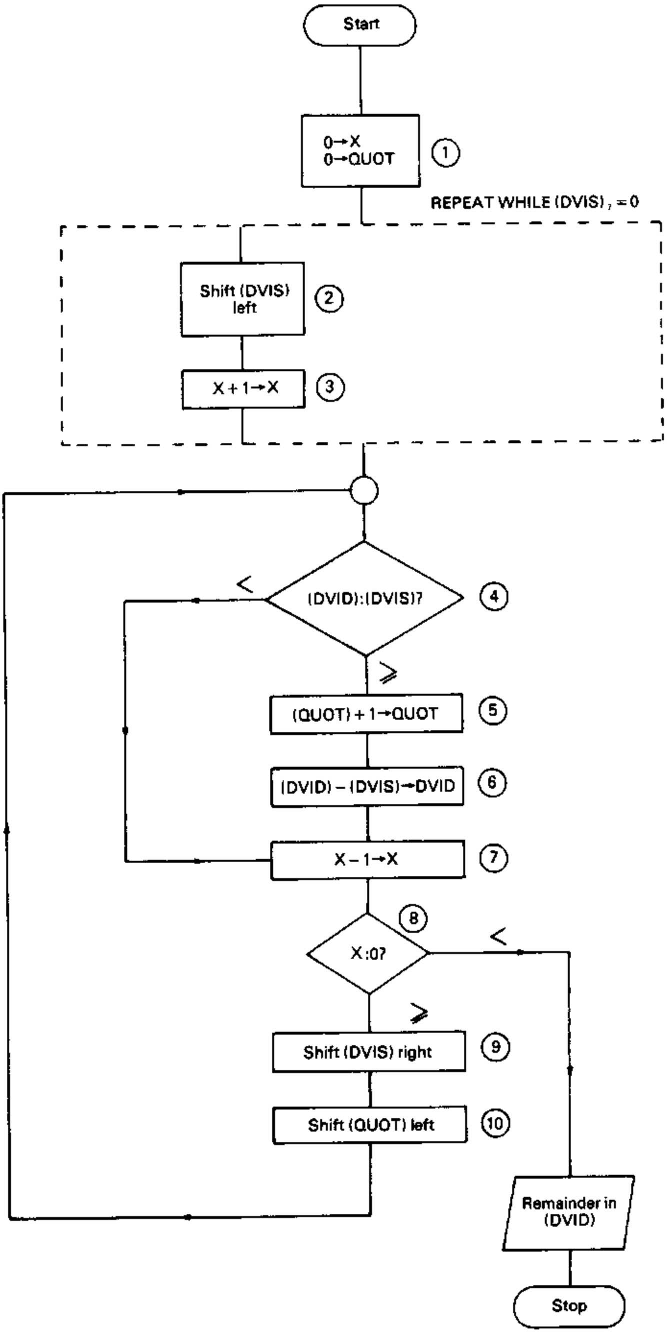
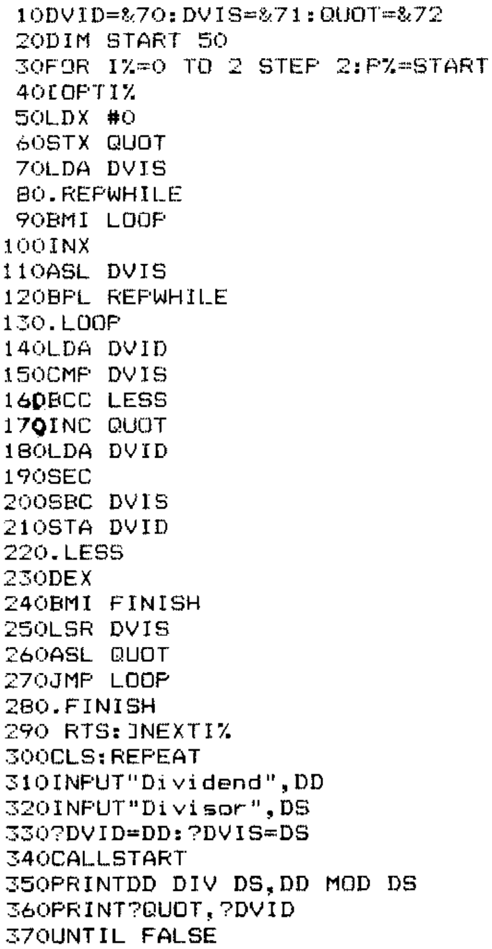

Start
0→X
0→QUOT
1
REPEAT WHILE (DVIS)₁ = 0
Shift (DVIS) left
2
X + 1→X
3
(DVID):(DVIS)?
4
<
⩾
(QUOT) + 1→QUOT
5
(DVID) – (DVIS)→DVID
6
X – 1→X
7
8
X :0?
<
⩾
Shift (DVIS) right
9
Shift (QUOT) left
10
Remainder in (DVID)
Stop

*Figure 8.6*

the REPEATWHILE loop: this is so we can test the N flag as it relates it DVIS; if we put INX after this, the N flag would refer to X.

LISTING 8.7



```
  10DVID=&70:DVIS=&71:QUOT=&72
  20DIM START 50
  30FOR I%=0 TO 2 STEP 2:P%=START
  40[OPTI%
  50LDX #0
  60STX QUOT
  70LDA DVIS
  80.REPWHILE
  90BMI LOOP
 100INX
 110ASL DVIS
 120BPL REPWHILE
 130.LOOP
 140LDA DVID
 150CMP DVIS
 160BCC LESS
 170INC QUOT
 180LDA DVID
 190SEC
 200SBC DVIS
 210STA DVID
 220.LESS
 230DEX
 240BMI FINISH
 250LSR DVIS
 260ASL QUOT
 270JMP LOOP
 280.FINISH
 290 RTS:]NEXTI%
 300CLS:REPEAT
 310INPUT"Dividend",DD
 320INPUT"Divisor",DS
 330?DVID=DD:?DVIS=DS
 340CALLSTART
 350PRINTDD DIV DS,DD MOD DS
 360PRINT?QUOT,?DVID
 370UNTIL FALSE
```

This program is very easy to extend to the division of a 16 bit dividend by a 16 bit divisor i.e. to any number from 0 to 32767 divided by any number from 1 to 32767. We need to assign two memory locations each to the divisor (DVIS+1; DVIS), the dividend (DVID+1; DVID) and the quotient (QUOT+1; QUOT). The program is left to you as an exercise in Exercise 8.3.

## Exercise 8.3

1. What happens in Listing 8.7 if we try to divide by zero? Include a line which will stop this happening.

2. Write the 16-bit program referred to in the section above; indicate a check for division by zero.

3. Write the code which will round the quotient to the nearest whole number in the program of Q.2. *Hint*: divide the divisor by 2 and round up if necessary; compare it with the remainder; unless the result is negative, add one to the quotient.

4. In a similar way to Q.3, Ex.8.1, write a program to divide two signed 8-bit numbers. Include a check for zero and a rounding operation, but do not output the remainder.

## 8.11 A SECOND APPROACH TO DIVISION

The programs developed in the last section are modelled closely on the structure of the pen-and-paper algorithm for division. It is worth examining, however, whether an improvement can be made by deviating from this structure a little.

Rather than shift the divisor, let us see what we can achieve if we instead shift the dividend. Consider again dividing an 8 bit dividend by an 8 bit divisor. The idea is that we shift the dividend left one bit at a time into the accumulator and at each stage compare the accumulator with the divisor. If the divisor is greater than or equal to the accumulator we subtract it from the accumulator and put a one in the least significant bit of the quotient. On each shift of the dividend we need also to shift the quotient one bit left.

Using the last example, the first five shifts left into the accumulator result in zeros in the quotient. The next shift has 101010 in the accumulator, so we subtract 11100, obtain 1110 in the accumulator and put 1 in the quotient which is now 000001. We now shift across the seventh bit to get 11101 in the accumulator; we subtract 11100 to obtain 1 in the accumulator and put 1 in the quotient which is now 0000011. We now shift the eighth bit giving 10 in the accumulator, which is the remainder. The quotient is 00000110, the final answer.

An examination of this algorithm indicates that we can save significant time and storage space by depositing the quotient in the same place as the dividend. Then a shift of the dividend also shifts the quotient as required. Returning to the last example, after the fifth shift the dividend will be 01000000. After the sixth we put a one in the least significant bit position to obtain 10000001; after the seventh we have 00000011; and after the eighth 00000110, as required.

A flowchart for the process is in Figure 8.7 and the program is in Listing 8.8. Notice how very much more concise it is than Listing 8.7. The moral of this is that we must choose our algorithm with great care: mapping exactly onto an existing pen and paper algorithm is not always the best course.

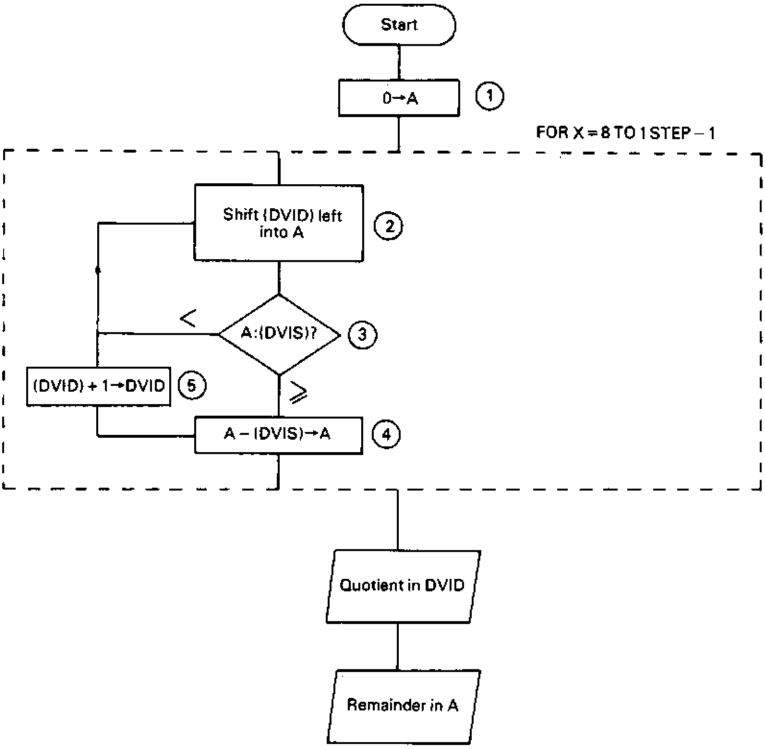


*Figure 8.7*

LISTING 8.8

```
   10DVID=&70:DVIS=&71:OSWRCH=&FFEE
   20DIM START 50
   30FOR I%=0 TO 2 STEP 2:P%=START
   40[OPTI%
   50LDA DVIS
   60BEQ MISTAKE
   70LDA #0
   80LDX #8
   90.LOOP
  100ASL DVID
  110ROL A
  120CMP DVIS
  130BCC LESS
  140SBC DVIS
  150INC DVID
  160.LESS
  170DEX
  180BNE LOOP
  190RTS
  200.MISTAKE
  210LDA #ASC("?")
  220JSR OSWRCH
  230LDA #7
```

```
240JSR OSWRCH
250RTS:]NEXTI%
260CLS:REPEAT
270INPUT"Dividend",A
280INPUT"Divisor",B
290?DVID=A:?DVIS=B
300!&403=USRSTART
310PRINTA DIV B,A MOD B
320PRINT?DVID,?&403
330UNTIL FALSE
```

Exercise 8.4

Write a program using this algorithm to divide a 16 bit dividend by a 16 bit divisor (use QUOTH for the higher byte of the shifted dividend and the accumulator for the lower). Is it shorter than that in Q.2, Ex.8.3? Is it quicker?

# Chapter 9 The stack: Subroutines and interrupts

## 9.1 THE CONCEPT OF A STACK

In Section 6.5 we met the idea of a queue. This was described as a FIFO structure, in that the first item to enter the queue was the first to be taken from it. The contrasting structure to this is the *stack,* for this is a FILO data structure. Thus, the *first* item on the stack is the *last* to be taken from it; and conversely, the last item on the stack is the first to be taken from it.

Now this data structure is much easier to implement than is the queue. First of all, we need have no worries about memory wastage; secondly we need only one and not two pointers.

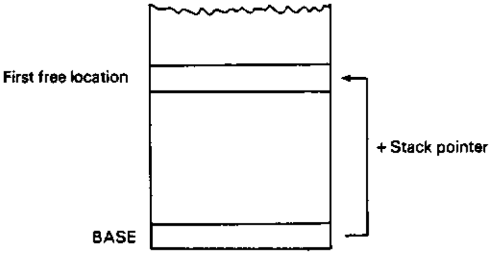


*Figure 9.1: The memory set-up for a stack.*

Figure 9.1 shows a section of memory to be used for a stack. Only two items of information are required: the base address of the stack (which is fixed) and the next free address in the stack.

Let us assume there are 256 bytes available for the stack. The first item to arrive is put at BASE + &FF, so the next free address is at BASE + &FE. The pointer to this address is called a *stack pointer,* and its initial value is &FF. As an item comes into the stack this pointer will be decremented by one. So after one item arrives, the stack pointer is &FE.

Notice that we have started to fill up the stack from the top: this is conventional. To complete this imagery, it is usual to refer to the putting of an item on the stack as *pushing* an item onto the stack. The image is best conceived as pushing the next item up a kind of tube, the base of which is at location BASE. We keep going until we get to the first free location.

When we want to get an item from the stack we *pull* it off, pulling it down the tube as it were.

Now all of this would be very easy to program in software. Let us use X for the stack pointer. We initialise the stack by writing LDX #&FF. Now, to put an item on the stack we write:

```
(a)                   STA BASE, X
                      DEX
```

assuming the item was originally in the accumulator.

And to take an item from the stack we write:

```
(b)                   INX
                      LDA BASE, X
```

Figure 9.2 shows the action of adding &10 to the stack when the pointer is &DA. Figure 9.3 shows the action of removing an item from the stack when the pointer is &E2.

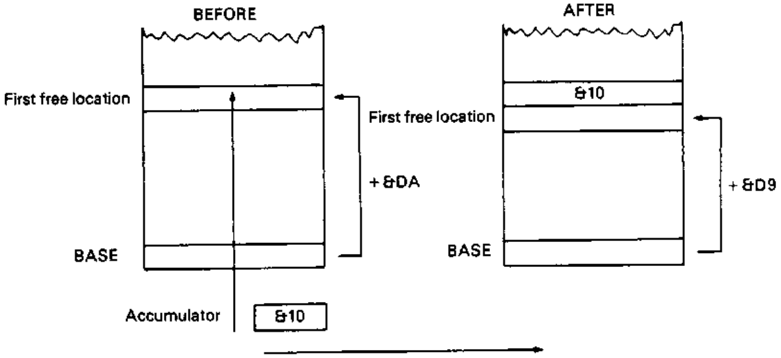


*Figure 9.2: Adding the item &10 to the stack.*

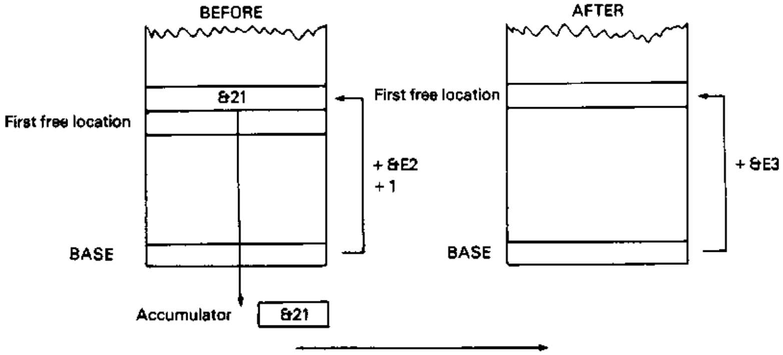


*Figure 9.3: Taking the next item (&21) from the stack*

Now the stack is such a useful device for temporarily storing information that the designers of the 6502 microprocessor have implemented the stack as part of its design. A special register, the stack pointer or SP, registers the next free location. And page one of memory, from &100 to &1FF is reserved for the stack (i.e. BASE = &100). Appendix 2 shows the SP register in relation to the rest of the architecture.

The useful thing about having this hardware stack (called a hardware stack because we do not have to write any special softwate to implement it) is that we do not tie up our X or Y registers in keeping track of it.

There are two special instructions to take the place of (a) and (b) above. Instead of (a) we write PHA: push the accumulator on the stack. The SP is automatically decremented. Instead of (b) we write PLA: pull the accumulator off the stack. The SP is automatically incremented.

When you turn the BBC computer on, the SP is automatically initialised to &FF. It is possible to do this yourself however; LDX #&FF : TXS is the coding. TXS transfers the contents of X to the stack pointer.

## 9.2 THE STACK AND NESTED SUBROUTINES

A particular program goes to a subroutine labelled SUBROUTINE1. Whilst in this subroutine another subroutine SUBROUTINE2 is called; whilst in SUBROUTINE2, SUBROUTINE3 is called; and then, SUBROUTINE4! When RTS is met, how does the microprocessor remember where to go back to? The answer lies in the stack.

Figure 9.4 shows the situation we have been discussing. The thin line shows the flow of logic as we call each subroutine in turn. The bold line shows the flow of logic as we return from each subroutine.

The first RTS (return from subroutine) which the microprocessor obeys is the one attached to SUBROUTINE4. At this stage the microprocessor needs to change the program counter to point to the next instruction after JSR SUBROUTINE4. The program counter increments uniformly, each new instruction being obeyed, until the next RTS is met. Now the microprocessor needs to change the program counter to point to the instruction immediately after JSR SUBROUTINE3; and so on, back to the instruction immediately after the first call at JSR SUBROUTINE1. How does the microprocessor do it?

The algorithm it uses is a very simple one: when it meets a JSR instruction it puts the current contents of the program counter onto the stack (*high* byte then *low* byte) before going off to the subroutine. 'Going off to the subroutine' means that the program counter takes on the address referred to by the label attached to JSR. When it meets an RTS, the microprocessor restores the address to the program counter by taking it off the stack.

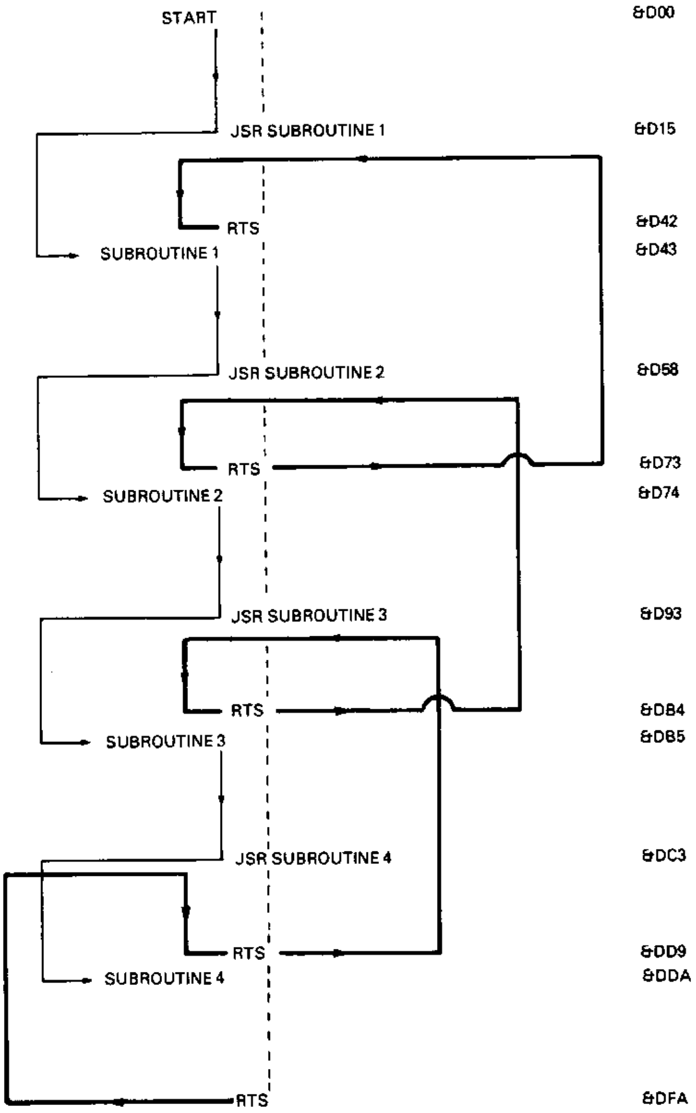


*Figure 9.4: Nested subroutines.*

Actually, there is a slight anomaly in the way the 6502 does this. The address it puts on the stack when it meets a JSR is in fact *one less* than that of the next instruction. Hence, when it meets an RTS it automatically *adds one* to the restored program counter before continuing.

To understand the full operation let us follow through the contents of the PC, SP and stack on moving through Figure 9.4. The memory locations are given on the right of that figure. Table 9.1 gives the details, where we assume (slightly unrealistically) that the stack begins empty.

| Present contents of program counter | New contents of program counter | Stack Pointer | Stack (first out on extreme left) |
|---|---|---|---|
| 0D00 | | FF | |
| 0D15 | 0D43 | FD | 17, 0D |
| 0D58 | 0D74 | FB | 5A, 0D, 17, 0D |
| 0D93 | 0DB5 | F9 | 95, 0D, 5A, 0D, 17, 0D |
| 0DC3 | 0DDA | F7 | C5, 0D, 95, 0D, 5A, 0D, 17, 0D |
| 0DFA | 0DC6 | F9 | 95, 0D, 5A, 0D, 17, 0D |
| 0DD9 | 0D96 | FB | 5A, 0D, 17, 0D |
| 0D B4 | 0D5B | FD | 17, 0D |
| 0D 73 | 0D18 | FF | |

*Table 9.1: The contents of PC, SP and stack during the operation of the program in figure 9.4.*

When we get to &D15, we meet JRS SUBROUTINE1. 0D, 17 goes onto the stack (first in is 0D so it is last out). Notice that &D17 is one less than the return address. The program counter now contains the address of SUBROUTINE1. At each new JSR the same thing occurs: onto the stack with the return address (less one); into the program counter with the address of the subroutine.

Then the first RTS is met at &DFA. The first byte off the stack goes into the low byte of the PC; the next into the high byte. Then the PC is incremented by one, and the execution of the program continues from &DC6, the next instruction after JSR SUBROUTINE4. And so it continues until we ultimately return to &D18.

You will notice a final RTS at &D42, but in Table 1 there is nothing left on the stack. In fact, there will be two more bytes: these point back into ROM, returning to the interpreter routine for CALL (USR is a little more complicated, but we need not consider it now). Thus, when we execute CALL&D00, a return address from the CALL routine is put on the stack, and the final RTS picks this up.

Apart from the CALL then, there is room for 126 nested subroutines in the 6502, (fewer with USR) far more than you are likely to need (unless you use a lot of recursion i.e. subroutines calling themselves). However, as we shall see later, the stack is also often used to store data using PHA and PLA.

The main purpose of using subroutines is to economise on program space. When using a high-level language, it is often advisable to use subroutines and procedures anyway, to make the program easier to follow. However, there are two disadvantages with subroutines which must be borne in mind if you wish to do this in assembly programming.

The first is that there is a time overhead to be paid. JSR takes 6 machine cycles to perform, and so does RTS, so there are at least 12 cycles in excess. Moreover, there may be a little more time taken up with passing parameters (see 9.4). On the BBC computer, 12 cycles is only 6 microseconds (6 × $10^{-6}$ secs) which may not seem very much. Indeed it isn't, but if your subroutine call is in a loop which executes a hundred thousand times (this is not untypical), then the subroutine call alone costs 0.6 secs, which is more substantial. An important principle, then, is not to use subroutines in large loops if time is critical.

The second disadvantage is that subroutines do not allow a program to be relocatable. By *reloctable,* we mean that the machine code translation can go into any part of memory and work, without having to be re-assembled. This means that there must be no JMPs (except for indirect jumps) and no JSRs (unless the subroutine goes to a fixed location like JSR OSWRCH). Branching is allowed, since it uses relative addressing, and in principle it is always possible to replace JMPs by branches, though it can sometimes be a little complicated. Relocatability is a desirable property, but not at the expense of program compactness or simplicity. Some programs are easy to make relocatable (an example is the REPLACE program, available on the second tape which can be purchased with this book; others are STRINGSORT and RETRIEVE in the next chapter). But in general, do not waste time and energy trying to do without subroutines just to achieve relocatability. With an assembler as fast as the BBC's, re-assembly only takes a moment anyway. However, there are sometimes useful tricks which can allow subroutines whilst retaining relocatable code: see Section 10.5 for an example of this.

## 9.3 INTERRUPTS

You are working at some fairly complex problem when the 'phone rings. You make a brief note of where you are up to, put aside your work and answer the 'phone. When you've finished, you get your work back, read the note you left yourself to remind you exactly where you were, and continue. The 'phone call represented an interruption, one too instrusive to ignore.

The microprocessor can be interrupted in a very similar way. One of its pins is connected to special circuitry such that, when this pin is grounded, an interrupt request passes to the microprocessor. The microprocessor finishes the instruction it is doing, saves some relevant information on the stack and goes off to an interrupt service routine. Figure 9.5 shows an example of this.

&D32 AD ←Interrupt happens while microprocessor here

&D33 03

&D34 04 ←The instruction (LDA &0403) is finished.

At this point, PC = &D35 and SR is, say, &81. Then,

&0D→stack

&35→stack

&81→stack

Address of interrupt service routine→PC.

*Figure 9.5: Program sequencing during an interrupt request.*

Now, sometimes you do not want the microprocessor to be disturbed in this way. You want to do something similar to taking the phone off the hook. You can do this by setting a flag in the processor status register, by writing SEI. This sets the interrupt disable flag. When you want the microprocessor to answer interrupts again, you enable interrupts by clearing the disable flag with CLI.

It is possible that some interrupts are just too important to ignore. For this reason, the 6502 has two separate interrupt lines: NMI for Non-Maskable Interrupt and IRQ for Interrupt Request. The former cannot be ignored, whether or not I is set; the latter will be ignored if I is set, but not if I is clear. The standard BBC microcomputer does not make any use of the NMI interrupt so we will have no more to say about it here.

When the microprocessor is interrupted, how does it know where the interrupt service routine is? The answer is that it always looks for the address of the routine in &FFFE and &FFFF. This happens automatically; it is part of the internal make-up of the 6502 and has nothing to do with the BBC machine. However, the address contained in &FFFE and &FFFF is at the disposal of the BBC micro's designers, and the code at that address must be provided by them. The address of the service routine is at &DDA4 for OS.01.

Now, there are a number of devices which could interrupt the microprocessor in the BBC computer: the tube, if it is connected; the printer or the user port, if available; the cassette system; and the analogue to digital converter, amongst others. However, the most usual interrupt comes from an internal timer which is part of a VIA (see Appendix 7), memory mapped from &FE40 to &FE4F.

You can disable interrupts from this timer only, by writing LDA #&40: STA &FE4E; but then no keyboard interruptions are possible, so JSR OSRDCH will not work, and neither will the input routines.

To re-enable interrupts from the timer, write LDA #&C0: STA &FE4E. You can disable *all* the interrupts with SEI; OSRDCH will still not work but entering the input routines will re-enable the flag.

Assuming no interrupts have been disabled, how can the microprocessor decide which device has caused the interrupt? The answer is that a number of memory locations need to be used to reflect which device is calling for service. For example, if bit 7 of &D5 is 1, this reflects the fact that the tube is asking for service; if bit 7 of &FE4D is 1, then it is the timer in the keyboard VIA which is asking for service. In this latter case, after the system clock has been updated, the routine will need to ascertain whether a key has been pressed on the keyboard too. The interrupt service routine must interrogate each of these status memory locations in a fixed order, and when it finds the cause of the interrupt it must service it.

It is conceivable that more than one device interrupts at the same time. In this case, more than one of the status locations will be set. The fixed order that the service routine uses to interrogate the status locations reflects the priority of the devices to be served. The tube gets a very high priority; the VIA timer gets a lower priority, which is one reason the system clock is not as accurate as the GMT signal, as John Coll puts it (another is that the crystal generators are not absolutely accurate, the error being about ± 0.1%, which is adequate for microprocessor timing, though not necessarily for clocks). Moreover, whilst one device is being serviced, it is essential that no other device can interrupt, otherwise some devices are in danger of never being serviced adequately. Because of this, IRQ is automatically kept low until the calling device is serviced. When the microprocessor returns from the interrupt, IRQ will go high again and new devices can interrupt.

When the microprocessor is interrupted, it automatically puts the return address on the stack (not less one as with a JSR). But in addition it puts the processor status register on the stack, because this register is very volatile (see Figure 9.5). Since there are three bytes to remove from the stack and not two, and since there need be no correction to the return address on the stack, RTS cannot terminate the interrupt. Instead, a new instruction RTI is used: this restores the processor status register, and then restores the return address (not incrementing it by one as in RTS).

Any service routine worthy of the name will certainly use the accumulator, and most will use the X and Y registers also. It is essential to preserve these registers, therefore, and very early in the service routine this must be done. We can save them on the stack by using PHA: TXA: PHA: TYA: PHA. At the end of the routine, just before the RTI, we restore the registers with PLA: TAY: PLA: TAX: PLA. Notice the reversal of order here when regaining the registers, having stored them in the order A, X, Y.

It is unlikely that you will want to interrupt the microprocessor in most of your programming, unless you are using the user port (on which see Appendix 7) but it is important to understand the general concept.

There is a software interrupt, called BRK, which you might be more tempted to use. The term 'software interrupt' needs to be explained first. The IRQ interrupt is a hardware interrupt, since it is produced by altering the voltage of a pin on the microprocessor. Sometimes, however, especially when debugging, one wishes to interrupt the flow of a program to go to some other routine. One way of doing this is to use BRK. This is a one byte instruction, and when working in machine code it makes it easy to patch in a software interrupt. At the place where we wish to stop we replace the next byte by &00, the Op Code for BRK. When the microprocessor encounters this, it puts onto the stack the return address *plus* one and the processor status register; and then goes to the software interrupt routine. Later, when we have completed the debugging, we can restore the old byte again.

The choice of zero as the Op Code for BRK is quite deliberate. When memory locations develop faults, they often give the impression of containing zero. Hence, by using a software interrupt routine in conjunction with BRK, faults of this type need not be catastrophic. But, the question arises: what is the address of the software interrupt routine? Where does the microprocessor get this information?

The answer is that it gets it from the same address as with IRQ; namely at &FFFE and &FFFF. This means that when it enters the interupt routine, it must check first to see if it is via BRK or IRQ. To enable it to do this, a special flag is reserved in the processor status register, the B flag. This is automatically set to 1 if a BRK is encountered. Hence, every service routine must start as follows:

```
STA TEMP

PLA

PHA

AND #&10

BNE BREAK
```

The accumulator is first saved temporarily. Then the processor status register is removed from the stack, and copied back to the stack. Then (line 4) bit 4 of the status register (the B flag) is isolated, so that the result will be zero if B = 0 and non-zero if B = 1. Hence, if the result is non-zero, we branch to the BRK servicing routine. Otherwise, we save all the registers on the stack, as mentioned earlier, and continue with the IRQ service routine.

Now the BRK command is used in a very special way by the BBC Computer. Most error messages are handled through the BRK command. What happens is that following BRK is put the error number. The address of this location is stored in &FD and &FE. A jump is then made to an address located at 0202, using the indirect jump instruction, JMP (&0202), which prints out the message located after BRK and the error number. Figure 9.6 shows the set-up for ESCAPE.

| | |
|---|---|
| 0 | BRK |
| 17 | Error number |
| 69 | E |
| 83 | S |
| 67 | C |
| 65 | A |
| 80 | P |
| 69 | E |
| 0 | Terminating sign (always zero). |

*Figure 9.6 The contents of memory locations designed to produce the message ESCAPE.*

Since &0202 is in RAM it is possible to create routines for oneself for BRK, but there are two problems here. Firstly, unless you incorporate a way of deciding whether BRK was encountered while checking your own assembly program rather than in the course of normal error processing in a BASIC program, you will lose all the error messages when using BASIC. For example, if you press ESCAPE (which is error number 17) and &0202 points to your own routine, you will jump into that routine. Even if you solve this problem, the processor status register has been removed from the stack and is lost to you. It follows that in the BBC Computer, BRK is not of much use for debugging. The next chapter will show you an alternative way to enlist a debugging aid into your program (Section 10.7).

Figure 9.7 acts as a summary to this section on interrupts, which has been fairly complicated.

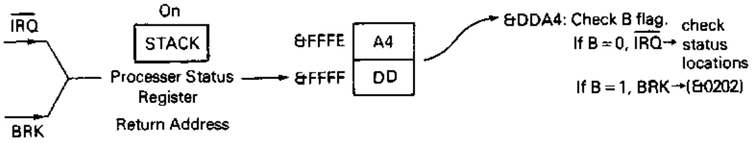


*Figure 9.7: Interrupts on the BBC Computer. ($\overline{IRQ}$ means that the IRQ pin on the microprocesser has been pulled low. BRK means that the opcode zero has been encountered in a program.)*

## 9.4 PASSING PARAMETERS TO AND FROM SUBROUTINES

There are three ways of passing information to and from subroutines, and we shall examine their merits here:

### (a) Through registers

This is the simplest way to convey information between the main program and the subroutine. If only one 8-bit parameter needs to be passed then the accumulator is the natural place: OSRDCH and OSWRCH use this method. If a second or third parameter is needed, X and Y can be used: OSBYTE uses this method.

If more than three parameters are to be passed, the registers can still be used as a pointer. A common convention is to put the low byte in X and the high byte in Y. Then the subroutine can store these in zero page locations of its choice and use indirect indexed addressing. This method is commonly used in operating system subroutines, where the firmware writer does not want to use up memory locations which will be used by a software programmer. For example, in the BBC Computer &70 to &8F are kept free: when one uses certain system calls that refer to a large memory allocation, one can specify what memory one wants to use by passing the pointer in X and Y. In this way, one can be free to partition one's own memory requirements and still have the advantage of using powerful system routines.

The disadvantage of using registers for passing parameters is that they might well contain important information which needs to be kept. The remedy here is to save the registers on the stack and recall them on return. It is good programming practice to save any registers used in the body of the subroutine, but not being used to pass parameters, in the subroutine itself. So, for example, if X and A are used to pass parameters, write PHA: TXA: PHA prior to loading X and A with the parameters and going to the subroutine. Y should be automatically preserved, for if the subroutine uses Y it will use TAY: PHA and recover Y with PLA: TYA at the end.

(b) Through the stack

Returning results from a subroutine on the stack or passing parameters to the subroutine using the stack needs care, since the return address is put on the stack just before entry and taken off just prior to exit. This means that the address needs to be saved in the subroutine itself. This is not generally used in firmware programs, but in your own programs it might be a possibility if you need to keep zero page free for some reason.

The first lines in the subroutine will be:

```
PLA
STA MEMLOC1 + 1
PLA
STA MEMLOC2 + 1
```

and the last lines will be:

```
MEMLOC2 LDA #0 Dummy operand
        PHA
MEMLOC1 LDA #0 Dummy operand
        PHA
        RTS
```

Notice the order of replacement: FILO dictates that the first item taken off be the last put on the stack.

Once this is done, items can be passed on the stack with complete freedom. If a lot of data is to be passed, this method could actually result in a saving of memory since PHA and PLA only take up one byte. The 14 bytes used up in storing and restoring the return address are a large overhead in most cases, however; and there is also a large time penalty attached.

I mentioned above that such techniques are not usually used in firmware. There is one exception to this, and this involves subroutines that may be called by the interrupt service routine, but which may also be used by the program being interrupted. In this case, it is conceivable that a section of program is interrupted, and then the interrupt routine uses just that section of program. In such a case, it is essential that the subroutines be capable of re-entry without destroying what has gone before. Such subroutines are called *re-entrant*. The fundamental consideration is that no fixed memory locations be used: this confines the subroutine to using registers (which are saved at the beginning of the interrupt routine) and the stack. If the stack is used, the X register will carry a zero page pointer to where the return address can be stored. A small section of zero page memory will be set aside for interrupt routine use, and its use will not corrupt any other program. The X register is used because it simplifies the coding. The subroutine will begin:

```
PLA
STA 0, X
PLA
STA 1, X
```

It is necessary that the X register is not otherwise used in the routine, since its contents will be required at the end when the return address is restored (unless there is room in zero page for this information also):

```
LDA 1, X
PHA
LDA 0, X
PHA
```

X is used in preference to Y here, since zero page addressing is not available with Y.

There is one other way that the stack can be used to pass data. Look at the following coding:

```
JSR EXAMPLE
M
E
S
S
A
G
E
0
0D
54
```

The word MESSAGE is encoded in ASCII following the JSR, and ends with zero. Following this, is the address to which the subroutine should return (minus *one*). The subroutine picks this up as follows:

```
EXAMPLE CLC
        PLA
        ADC #1
        STA ADDRESS
        PLA
        ADC #0
        STA ADDRESS+1
```

where ADDRESS is a zero page location. Now, indirect indexed addressing can be used to pick up the message i.e. LDA (ADDRESS), Y. When zero is encountered, the message is over and the subroutine returns as follows:

```
LDA (ADDRESS), Y
PHA
INY
LDA (ADDRESS), Y
PHA
RTS
```

This method is not re-entrant, and so is not used in interrupts, but messages are not usually needed in interrupts in any case. The BBC Computer uses this method in cases where a message is required but the BRK method, discussed earlier, is not suitable.

### (c) Through fixed memory locations

This is the method that you will use most often when you write your own subroutines. It is generally frowned upon by many authorities on programming because it makes it difficult to establish a library of subroutines. This is because a subroutine may use memory which interferes with the main program.

This is not a problem on the BBC Computer, however, as long as you store a copy of your program in its unassembled (source code) form. It can then be combined with other subroutines and the main program, and memory allocated at the end. Appendix 6 explains how to combine such programs together.

The only problem with this method is that labels may interfere with each other. For example, we may have OVER in the main program and the subroutine. It should not be difficult, however, to check a program for this. A utility program on the second tape (FINDCODE) will be of assistance in this.

## 9.5 TWO IMPORTANT SUBROUTINES

In order to illustrate the ideas considered in the last section, especially those in (c), let us examine two very useful subroutines. The first accepts numerical input in decimal from the keyboard and converts it into a signed four byte hex number. It therefore fulfils the function of INPUT N as opposed to INPUT N$, which we covered in Chapter 6. The second outputs in decimal a signed four byte hex number: it fulfils the function of PRINT N.

LISTING 9.1

(a) A subroutine to convert from decimal to hex

```
 10LOOPCOUNT=&70:INDIC=&71:BUFFER=&73:SPOINT=&75:NUMBER=&76:
   OSWRCH=&FFEE
 20DIM START 200
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50PHA
 60TXA
 70PHA
 80TYA
 90PHA
100TSX
110STX SPOINT
120LDA #ASC("?")
130JSR OSWRCH
140JSR &BC20
150LDY #0
160LDX #3
170.INITIALISE
180STY NUMBER,X
190DEX
200BPL INITIALISE
210STY INDIC
220STY BUFFER
230LDA #7
240STA BUFFER+1
250LDA (BUFFER),Y
260CMP #ASC("-")
270BNE NTMINUS
280LDA #&80
290STA INDIC
300BNE OVER
310.NTMINUS
320CMP #ASC("+")
330BNE CHARGET
340.OVER
350INC BUFFER
360.CHARGET
370LDA (BUFFER),Y
380CMP #&0D
390BEQ CHECKSIGN
400JSR MULTTWO
410LDX #3
420.STACK
430LDA NUMBER,X
```

```
440PHA
450DEX
460BPL STACK
470JSR MULTTWO
480JSR MULTTWO
490LDX #0
500LDA #4
510STA LOOPCOUNT
520CLC
530.XTEN
540PLA
550ADC NUMBER,X
560STA NUMBER,X
570INX
580DEC LOOPCOUNT
590BNE XTEN
600BVS OVERFLOW
610LDA (BUFFER),Y
620SEC
630SBC #&30
640BCC MISTAKE
650CMP #10
660BCS MISTAKE
670ADC NUMBER
680STA NUMBER
690BCC NOCARRY
700INC NUMBER+1
710BCC NOCARRY
720INC NUMBER+2
730BCC NOCARRY
740INC NUMBER+3
750BMI OVERFLOW
760.NOCARRY
770INY
780BNE CHARGET
790.CHECKSIGN
800LDA INDIC
810BPL PLUS
820JSR FLIPSIGN
830.PLUS
840CLC
850BCC REGS
860.OVERFLOW
870LDA INDIC
880ORA #&40
890STA INDIC
900.MISTAKE
910LDX SPOINT
920TXS
930SEC
940.REGS
950PLA
960TAY
970PLA
980TAX
990PLA
```

```
1000RTS
1010.MULTTWO
1020ASL NUMBER
1030ROL NUMBER+1
1040ROL NUMBER+2
1050ROL NUMBER+3
1060BMI OVERFLOW
1070RTS
1080.FLIPSIGN
1090LDY #4
1100LDX #0
1110SEC
1120.FLIP
1130LDA #0
1140SBC NUMBER,X
1150STA NUMBER,X
1160INX
1170DEY
1180BNE FLIP
1190RTS:]NEXTI%
1200REPEAT
1210!&404=USRSTART
1220IF (?&407 AND 1) = 0 THEN PRINT!NUMBER ELSE IF
    (?INDIC AND &40) = 0 PRINT"Not a valid number" ELSE
    IF (?INDIC AND &80) = 0 PRINT"Number too large" ELSE PRI
    NT"Number too small"
1230UNTIL FALSE
```

Listing 9.1 contains the program and the details are:

50-90 Saves registers on the stack.

100-110 Save stack pointer in case exit from program is via OVERFLOW (at line 870) or MISTAKE (at line 910). TSX transfers a copy of the stack pointer to the X register.

120-140 Display ? prompt, and get data from keyboard into buffer at &700.

150-240 Initialise the relevant memory locations.

250-300 If the first character is a minus sign, set bit 7 of INDIC to 1. Line 300 always results in a branch to 350.

320-330 If the first character is not a plus sign, skip to 370.

350 If first character was a +/- sign, increment buffer to begin at &701.

370-390 If end of number (signalled by carriage return), go to check the sign at line 800.

400-460 Otherwise, multiply the current contents of the four byte location NUMBER by 2 and save on the stack with least significant byte last in (and so first out in line 540).

470-480 Multiply NUMBER by 4 more, giving a total multiplication of 8.

490-590 Add the four bytes on the stack (deposited in 400-460) to the current contents of NUMBER. This is equivalent to 2 * NUMBER + 8 * NUMBER i.e. it multiplies NUMBER by ten.

600 If overflow occurs into bit 31, jump to error exit at 870.

610-660 Load the current buffer character into the accumulator, and subtract &30 to convert to a digit from 0 to 9. If the result is not 0 to 9, go to error exit at 910.

670-740 Add this digit to the current contents of NUMBER. (Note that CLC is unnecessary since C must be zero to get past line 660.)

750 As line 600, but overflow into bit 31 is signalled after INC by the N flag.

770-780 Increment the pointer to the next character in the buffer, and branch always to line 370.

790 This section alters the sign of the result if required.

800-820 If (INDIC)$_7$ = 1, convert NUMBER to its negative counterpart at line 1090.

840-850 C = 0 will indicate 'no error' to the main program. Always branch to line 950.

860-890 This is entered if overflow has occurred. Set (INDIC)$_6$ to 1 to indicate 'overflow error' to main program.

910-920 If any error has occurred, restore stack pointer to the old value on entry, just after the registers were saved. This is a precaution in case an error was encountered while the stack still contained some data for the conversion (e.g. at line 1060).

930 Set C = 1 to indicate 'error'.

950-1000 Restore registers (note the reverse order from 50-90) and return.

1010-1060 A subroutine to multiply the four byte NUMBER by 2, and to check for overflow.

1080-1190 A subroutine to reverse the sign of NUMBER, leaving its numerical value unchanged. The method used is to subtract from &00000000. This is quicker than using EOR #&FF on each byte plus CLC: ADC #1 at the end.

Notice that we cannot use CPX #4 since this affects carry. So we have a second counter in Y which goes from 4 to 0 using DEY, since this does not affect carry.

The general algorithm is very simple: begin with NUMBER equal to &00000000. Then continue to multiply by ten and add the next digit (0 to 9) in the buffer until carriage return is met. For example, 312 is evaluated as ((0 × 10 + 3) × 10 + 1) × 10 + 2.

If at any point an error occurs, we clear the stack of any unwanted data, signal the error and return. The possible errors are 'invalid number' (e.g. 31C2) or overflow (e.g. 10,000,000,000).

An error is signalled to the main program which calls this as a subroutine, by setting the Carry flag. Here, we use BASIC to test the program in lines 1200-1230. Notice that we are able to differentiate between the type of error, and also differentiate between positive and negative overflow.

The carry flag is the best flag to use for indicating error: it is easy to set and to clear (SEC and CLC). It is also very easy to test: BCS Error. If a second flag is required, the overflow is best, since this can easily be cleared with CLV. How we set it will be discussed in a moment. If a third flag is required use the N flag: we will consider in a moment how to set and clear this.

Now in this program, we only need to use one flag to indicate error, since we already have the location INDIC at our disposal to convey the other information. But how do we discover the contents of bits 6 and 7 of INDIC? Bit 7 seems straightforward: just use LDA INDIC: BPL ...., but what about bit 6, which we need to test first in this case?

There is an instruction BIT which will do everything for us. Just writing BIT INDIC sets the N flag to bit 7 of INDIC and the V flag to bit 6. Moreover, it sets the Z flag to 1 if "ANDing' the accumulator and INDIC (i.e. A∧(INDIC)) is zero; Z is set to zero otherwise. This last use of BIT is not needed here, but it is useful when we wish to test certain bits of a whole series of locations. The reason for this is that BIT leaves the accumulator unchanged, whereas AND changes the accumulator. In a way, BIT is a sort of logical equivalent to CMP. So, if we wanted to see if either bits 2 or 4 of any of four locations were one we could write: LDA #&14: BIT LOC1: BNE ONE: BIT LOC2: BNE TWO etc. This is particularly useful in interrupt processing.

However, the main use of BIT is to access bits 6 and 7 of a location easily and moreover without affecting the accumulator. A main program calling this subroutine would first test C. If C is 1 it goes to 'Error' where the following will occur: BIT INDIC : BVC INVALCHAR : BPL TOOBIG : ..... If bit 6 of INDIC is zero, INVALCHAR will print a message like 'Not a valid number'; otherwise if bit 7 of INDIC is zero, there is positive overflow, and TOOBIG will print 'Number too large'. If bit 7 is 1, we pass through to some code which prints 'Number too small'. After this, the main program will return to the subroutine to try again. In this way, errors are easily dealt with.

It is important to signal errors in this way, preferably using flags unless memory locations are available, as here. If we wish to use V and perhaps N this is how we can set and clear them:

| | | |
|---|---|---|
| SET N and V | BIT &FFFB | (&FFFB contains &FF) |
| SET V, clear N | BIT &FFF7 | (&FFF7 contains &6C) |
| SET N, clear V | BIT &FFFA | (&FFFA contains &A0) |
| CLEAR N and V | BIT &FFF9 | (&FFF9 contains &02) |
| CLEAR V, N immaterial | CLV | (N will be unchanged by this) |

In all of these, the contents of the accumulator will be unaffected. When return is made to the main program, it is important that tests are made immediately if N is being used, since it is extremely sensitive to almost every instruction.

One last point on setting flags. Is it possible to set V leaving N unchanged? It is, but to do it we need to introduce two new instructions: PHP and PLP. PHP copies the processor status register to the stack; PLP copies the top item on the stack to the processor status register. So we write:

```
PHP : PLA : ORA #&40 : PHA : PLP
```

Transfer the status register to the accumulator. Put a 1 into bit 6, the V flag, leaving all other bits unchanged. Transfer the accumulator back to the status register. All transfers between P and A have to be done via the stack.

Appendix 6 explains how to append Listing 9.1 to another program. The listing will need to be on tape, renumbered from 30,000 onwards: lines 10-40, 1190-1230 will not be required. The program will end with RTS and line 30,000 will give a list of all the variables used (i.e. it will be line 10 without the location assignments). The main program can now fix memory locations for these variables, and a quick check can be made to ensure that no labels are used twice. To help with this, it is best if common labels like LOOP, NOCARRY etc. are used only in main programs. In the case of NUMBER, however, it is probably preferable to use this variable name in the main program too, to facilitate the easy passing of parameters. Line 30,000 can then be deleted, and the entire program renumbered.

If subroutines are always saved with at least one copy numbered from 30,000, and if no main program has a greater line number than 30,000, then it is easy to append as many subroutines as required, since after appending, renumbering can be used and the next subroutine appended (32767 is the highest line number allowable).

(b) A subroutine to convert from decimal to hex

LISTING 9.2

```
 10NUMBER=&70:OSWRCH=&FFEE:OSNEWL=&FFE7
 20DIM START 100
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50PHA
 60TXA
 70PHA
 80TYA
 90PHA
100LDA NUMBER+3
110BPL PLUS
120LDA #ASC("-")
130JSR OSWRCH
140JSR FLIPSIGN
150.PLUS
160LDX #0
170.CONVERT
180LDY #32
190LDA #0
200.TENDIV
210ASL NUMBER
220ROL NUMBER+1
230ROL NUMBER+2
240ROL NUMBER+3
250ROL A
260CMP #10
270BCC LESS
280SBC #10
290INC NUMBER
300.LESS
310DEY
320BNE TENDIV
330CLC
340ADC #&30
350PHA
360INX
370LDA NUMBER
380ORA NUMBER+1
390ORA NUMBER+2
400ORA NUMBER+3
410BNE CONVERT
420.DISPLAY
430PLA
440JSR OSWRCH
450DEX
460BNE DISPLAY
470JSR OSNEWL
480PLA
490TAY
500PLA
510TAX
520PLA
530RTS
540.FLIPSIGN
550LDY #4
560LDX #0
570SEC
580.FLIP
590LDA #0
600SBC NUMBER,X
610STA NUMBER,X
620INX
630DEY
640BNE FLIP
650RTS:]NEXTI%
660REPEAT
670INPUT"Number",!NUMBER
680CALLSTART
690UNTIL FALSE
```

Listing 9.2 contains the program and the details are:

50-90 Save registers on the stack.

100-140 If number is negative, output a minus sign and make the number positive (at line 550).

160 Set digit counter to zero.

180-190 Set bit counter to 32 and initialise the accumulator.

210-320 Divide current contents of the four byte NUMBER by 10, using the method described in Section 8.9.

330-350 The remainder is in the accumulator, and this digit is the next digit to be output, reading from right to left. Convert to ASCII and save on the stack.

360-410 Increment digit count. NUMBER now contains the result of dividing by 10 in lines 210-320. If NUMBER is not zero, go back and divide by 10 again to get the next digit.

430-470 Output digits from the stack. They will now be in the correct order, since the last one in was the digit furthest to the left.

480-530 Replace registers and return.

550-650 As 1090-1190 in Listing 9.1.

The algorithm used here is again simple: keep dividing by 10, saving the remainders on the stack. Output them in reverse order when the dividend becomes zero. For example, applying this to &BC7 gives: remainder 5, dividend &12D; remainder 1, dividend &1E; remainder 0, dividend 3; remainder 3, dividend 0. Now restoring the remainders in reverse order gives 3015.

Exercise 9.1

1. Use PHP and PLP to set or clear N and V without affecting the Z flag.

2. Use PHP and PLP to perform the same function as lines 260-300 of Listing 7.7, but dispensing with the X register. Which method is more efficient?

## 9.6 FURTHER USES OF THE STACK

We have seen in the last section two uses of the stack to store temporary information. In the first, it was convenient to put the interim result of 2 * NUMBER on the stack. We could have saved it temporarily in memory but it would have taken up zero page memory.

In the second example, not only did the stack act as a temporary storage area but it also reversed the data for us into the correct order. This is an important use of the stack.

There have been many occasions throughout this book where the instruction STA TEMP has been used, with LDA TEMP following some time later. In such cases PHA and PLA would have saved space. Consider the answer to the first part of Q.3 of Ex.5.3 again. Using the stack we could write instead:

```
PHA : TXA : SEC : SBC M : TAX : PLA
```

This is more economical: it also solves the problem for a program written in ROM.

LISTING 9.3

```
 10COLUMNS=&70:ROWS=&71:SCREENLEFT=&72:SCREENRIGHT=&74:
   TEMP=&76:OSBYTE=&FFF4
 20FOR I%=0 TO 2 STEP2
 30P%=&D00
 40[OPTI%
 50LDX #4
 60LDA #&85
 70JSR OSBYTE
 80STY MEMLOC+1
 90LDA #&84
100JSR OSBYTE
110STX SCREENRIGHT
120STY SCREENRIGHT+1
130.MEMLOC
140CPY #0 Dummy operand
150BNE ZEROMODE
160LDA #39
170BNE FOURMODE
180.ZEROMODE
190LDA #79
200.FOURMODE
210STA COLUMNS
220LDA #32
230STA ROWS
240.BEGIN
250LDY #7
260.LOOP1
270LDA (SCREENRIGHT),Y
280PHA
290DEY
300BPL LOOP1
310LDX COLUMNS
320.LOOP2
330LDA SCREENRIGHT+1
340STA SCREENLEFT+1
350LDA SCREENRIGHT
360STA SCREENLEFT
370CLC
380ADC #8
390STA SCREENRIGHT
400BCC NOCARRY1
410INC SCREENRIGHT+1
420.NOCARRY1
430LDY #7
440.LOOP3
450LDA (SCREENRIGHT),Y
460STA (SCREENLEFT),Y
470DEY
480BPL LOOP3
490DEX
500BNE LOOP2
510LDY #0
520.LOOP4
530PLA
540STA (SCREENRIGHT),Y
550INY
560CPY #8
570BNE LOOP4
580LDA SCREENRIGHT
590CLC
600ADC #8
610STA SCREENRIGHT
620BCC NOCARRY2
630INC SCREENRIGHT+1
640.NOCARRY2
650DEC ROWS
660BNE BEGIN
670RTS:]NEXTI%
680time%=TIME
690FOR I%=1 TO 100:CALL&D00
700NEXTI%:time%=TIME-time%:PRINTtime%/10000 - .001;"secs
   per shift left"
```

Listing 9.3 gives one more example of the use of the stack. This program rotates the high-resolution screen one character to the left. Refer to Section 8.6 for the discussion on high resolution graphics organisation, if you need to. The details are:

50-80 As 80-110, Listing 8.6 (Y stored at line 140).

90-120 As 120-150, Listing 8.6, except we store starting address in SCREENRIGHT.

140-150 As 170-180, Listing 8.6.

160-170 Put the number of columns less one of mode 4 into the accumulator and branch to 210.

190 The number of columns less one for mode 0.

210 Store number of columns less one in COLUMNS, a permanent location.

220-230 Number of rows in ROWS.

250 Initialise byte counter for left-most character square (8 rows, one byte per row).

270-300 Put the first character on the left of the current row on the stack.

310 Initialise column counter.

330-410 SCREENRIGHT refers to the address of the right-most character of the current pair and SCREENLEFT the left-most. This section of code increments these addresses by 8, so that they point to the next pair.

430 Byte counter for current character square.

440-480 Move the right-most character of the pair into the left-most character position.

490-500 Continue for the 79 or 39 columns (not including the last column).

510 Byte counter for right-most character square.

530-570 Get left-most character square from the stack and store in right-most character position.

580-630 Increase SCREENRIGHT to the address of the beginning of the next row.

650-670 Continue for 32 rows, then return.

The idea is to store the first character of the row on the stack (in lines 270 to 300), shift all the rest one place to the left, and then restore the contents of the stack into the last character position (in lines 530-570). Notice how the reversing property of the stack is a slight inconvenience here: it necessitates a forward loop with a CPY at 560. Nevertheless, the stack saves memory here but not time. Why this is so we will consider in the next section.

## Exercise 9.2

1. Rewrite 1(i) and 2 of Ex.5.3 so that they would work in ROM.

2. Rewrite the second part of Q.3 and Q.4, using the stack as much as possible.

## 9.7 TIMING

Appendix 1 contains information on the timing for each instruction. For example, PHA takes 3 cycles, the same as STA TEMP, where TEMP is in zero page. PLA takes 4 cycles, one more than LDA TEMP. It is therefore slightly quicker, on average, to use zero page memory than the stack, but there is a memory cost both in zero page and in the program itself (PHA and PLA are both one byte instructions). Moreover, there can be a time saving using the stack. In Listing 9.3, if we used zero page memory in 270-300 instead of the stack, we would need to use indexed addressing (STA TEMP, Y): this takes five cycles, two more than PHA. However, to be fair there is a corresponding saving in 530-560: we can move backwards through the loop, by replacing PLA by LDA TEMP, Y and can thus eliminate CPY #8. This saves 2 cycles altogether (CPY #8 is 2 cycles, PLA and LDA TEMP, Y are both 4 cycles), which cancels out the extra two in the temporary storage loop.

You will have noticed that the average time taken per shift in mode 4 is about 0.1067 seconds in Listing 9.3. The 6502 microprocessor in the BBC computer runs at 2 MHz i.e. 2 million cycles per second. It follows that Listing 9.3 uses up about 200,000 machine cycles! Where do they all go?

In order to answer this we will analyse the timing involved in Listing 9.3. Table 9.2 gives the details. The column labelled multiplication factor, gives the number of times the instruction is used in the program. In the most used loop, LOOP3, each instruction is used 10,240 times! This underlines the importance of making loops as economical as possible.

In certain cases we have had to estimate the factor: line 630 is an example. With branches, there is a problem since if the branch occurs, the time involved is 3 instead of 2 cycles. We have used the method of assigning the time of the most frequent occurrence - the results should then average out. In this case all branches are given the value of 3: this will involve only a slight over-estimate overall (we  take no account of page crossing, which gives an additional cycle to a branch, and to indexed addressing, since it will not occur in this program - this is why we have

| Line number | Time for instruction | Multiplication factor | Total |
|---|---|---|---|
| 50 | 2 | 1 | 45 |
| 60 | 2 | 1 | |
| 70 | 6 | 1 | |
| 80 | 4 | 1 | |
| 90 | 2 | 1 | |
| 100 | 6 | 1 | |
| 110 | 3 | 1 | |
| 120 | 3 | 1 | |
| 140 | 2 | 1 | |
| 150 | 2 | 1 | |
| 160 | 2 | 1 | |
| 170 | 3 | 1 | |
| 190 | 2 | 0 | |
| 210 | 3 | 1 | |
| 220 | 2 | 1 | |
| 230 | 3 | 1 | |
| 250 | 2 | 32 | 64 |
| 270 | 5 | 32 × 8 | 3328 |
| 280 | 3 | | |
| 290 | 2 | | |
| 300 | 3 | | |
| 310 | 3 | 32 | 96 |
| 330 | 3 | 40 × 32 | 28160 |
| 340 | 3 | | |
| 350 | 3 | | |
| 360 | 3 | | |
| 370 | 2 | | |
| 380 | 2 | | |
| 390 | 3 | | |
| 400 | 3 | | |
| 410 | 5 | 40 | 200 |
| 430 | 2 | 40 × 32 | 2560 |
| 450 | 5 | 40 × 32 × 8 | 163840 |
| 460 | 6 | | |
| 470 | 2 | | |
| 480 | 3 | | |
| 490 | 2 | 40 × 32 | 640 |
| 500 | 3 | | |
| 510 | 2 | 32 | 64 |
| 530 | 4 | 32 × 8 | 4352 |
| 540 | 6 | | |
| 550 | 2 | | |
| 560 | 2 | | |
| 570 | 3 | | |
| 580 | 3 | 32 | 416 |
| 590 | 2 | | |
| 600 | 2 | | |
| 610 | 3 | | |
| 620 | 3 | | |
| 630 | 5 | 6 | 30 |
| 650 | 5 | 32 | 256 |
| 660 | 3 | | 6 |
| 670 | 6 | 1 | |
| | | Total | 204,057 |

*Table 9.2: Timing for listing 9.3*

begun at &D00 rather than use DIM START 150). In cases where exactness is important (we shall meet one in a moment) we shall need to be more accurate in our allocation.

From our analysis we see that over 80% of the time is spent on the loop in lines 450-480! Unfortunately, indirect indexed addressing is used twice in this loop and this is costly in time (the reason for this difference in time would take us into hardware considerations outside the scope of this book). But even if we could have got away with zero page indexed addressing, we would save only 3 cycles per loop, which would result in an overall saving of 15%. And of course in mode 0, the time is very nearly doubled.

This sort of analysis does not need to be done with many programs, but it is very revealing in this case. There is a general principle stemming from this: worry about saving time only in loops; initialisations and very small loops take up a negligible amount of time anyway, and are not worth the time-saving effort.

Even with such economy, this rotation program is rather slow for some purposes: for greater speed, the properties of the 6845 chip and screen ULA would have to be used, but these are outside our scope.

To end this section let us try to devise a piece of code which will result in a pause of *precisely* 1 millisecond (i.e. 2000 machine cycles). In doing so, we will introduce our last new instruction: NOP. This instruction does nothing: it just takes up 2 cycles of processor time, and is useful for the fine tuning of timing loops, as we shall see.

To begin with we must set the interrupt disable: the interrupt routine takes time and will muddle our timing. We will not clear the disable as part of the code, since that would be done after the operation for which the pause was required has been completed.

Consider the loop

```
LOOP DEX
     BNE LOOP
```

This takes up 5 * X -1 cycles (minus 1 since the last branch will not occur). The maximum value of this is 1279, which falls below our 2000 mark.

Consider instead

```
LOOP DEC DELAY
     BNE LOOP
```

This takes up 8 * (DELAY) - 1 cycles assuming zero page. We are in range here, so let us now consider the initialisation. We have:

```
SEI
PHP
PHA
```

```
          LDA #Count
          STA DELAY
```

and at the end we will need PLA : PLP.

We do not want our delay loop to affect any of the workings of the rest of the program, so we save the accumulator and status register on the stack. All of this adds up to 21 cycles: we therefore require 8 * Count - 1 = 1979, and the nearest we can get to this is Count = 247, which leaves us 4 cycles to find. This is where NOP comes in. So our 1 millisecond delay is:

```
     SEI
     PHP
     PHA
     LDA #247
     STA DELAY
LOOP DEC DELAY
     BNE LOOP
     NOP
     NOP
     PLA
     PLP
```

Exercise 9.3

Show that if we wish to generalise this to a t millisecond delay by using the X register containing t, then, assuming we will save the X register on the stack, we will need to load DELAY with the greatest integer not more than (1991t - 28)/8t.

Include suitable code to give a precise pause of 10 milliseconds.

## 9.8 SCREEN SCROLLING: HOW IT OPERATES

If we consider again the operation of Listing 9.3, something rather odd will be noticed. To slow things down a little, insert a line 695 F = INKEY(25) in the program.

Clear the screen in mode 4, begin listing the program and press escape before any scrolling occurs. Now type RUN, and you will notice the screen shuffling along from the top. Now type LIST again, and after a few scrolls press escape. Type RUN again, and you will notice that the screen shuffles along again, but this time it begins somewhere around the middle. Remove line 695 and repeat these experiments: you will see the same effects happening more quickly. In the second case, some of the lines will seem to be indented as the rotation occurs.

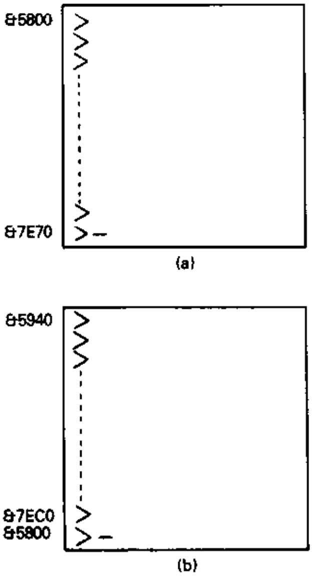
To understand what is happening here, we need to perform one more experiment. Keeping Listing 9.3, add the following line:

```
    1      FOR I% = HIMEM TO &7FFF: ?I% = &AA: F = INKEY(5): NEXT
```

(use &3FFF in a model A)

Clear the screen and type RUN. The screen will begin filling up from the top. Press escape, and list the program to get some scrolling. Type RUN again and watch the screen fill up starting at the middle! Now clear the screen again, list a little without scrolling, and press Return enough times to put the cursor at the bottom of the screen. Now type RUN, and one scroll will occur. The screen will begin to fill up from the bottom. Try this again, but with one scroll before typing RUN and the next to bottom line will start filling; with two scrolls plus RUN, it begins filling two lines from the bottom; and so on.

It is scrolling which causes this to happen. Let us consider what happens in scrolling. Imagine that the whole screen is full and that the cursor is now at the bottom of the screen. At this stage we type in something and press return: what must happen now is that the line at the top of the screen disappears to accommodate the new flashing cursor line. One way to do this is to move up all



Before pressing return, start of screen is at &5800 (assuming we began with a clear screen).

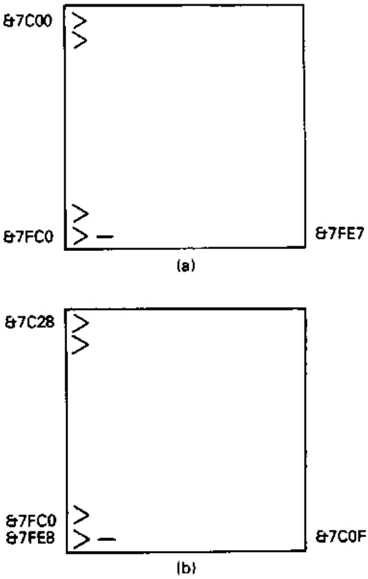
After pressing return, start of screen is now at &5940. The contents of &5800 are now replaced by the flashing cursor line at the bottom of the screen.

*Figure 9.8: Scrolling in mode 4 (model B addresses)*

the other lines so that line two is now at HIMEM (i.e. &1800 or &5800), but this is rather slow. A simpler approach is to replace the contents of the top line on the screen by the flashing cursor line (i.e. > followed by blanks). All we need to do then is to redefine the start of the screen as the location of the present line one (which is the old line two) i.e. &5940 (or &1940) in Mode 4. It follows then that &5800 (or &1800) is now the location for the bottom line of the screen. Figure 9.8 shows this.

The screen memory is hence treated as a sort of cylinder, rather like our method in 6.5.when we implemented the queue. All modes are treated like this, not only Mode 4.

The situation with Mode 7 is slightly different, since there are 24 unused locations in screen memory. As scrolling takes place, the 24 locations which are unused will change. Figure 9.9 illustrates this. Here, the cylinder principle is even more obvious. Similar considerations apply to Modes 3 and 6 also.



Before pressing return, start of screen is at &7C00 (assuming we began with a clear screen). The 24 locations &7FE8 to &7FFF are unused.

After pressing return, start of screen is now at &7C28. The bottom line begins at &7FE8, at the 24th character reaches &7FFF, and then returns to &7C00 for the 25th character. The 24 locations from &7C10 to &7C27 are unused.

*Figure 9.9: Scrolling in mode 7 (model B addresses.)*

There are a pair of memory locations which are set aside to hold the information of where the start of the screen is at any particular time: the low byte is in &322 and the high byte in &323. Incidentally, the figure given for each mode is the same whether or not a model B is in use. So, a clear screen in Mode 7 gives &7C00 for both model A and model B. However, the BBC Computer copes with this in a rather novel way: in a model A machine, all references to location &4000 to &7FFF are treated as references to &0 to &3FFF. In this way, the computer can store one set of parameters in ROM which will work for either machine.

### Exercise 9.4

Rewrite Listing 8.6 so that it will cope with scrolling.

# *Chapter 10 Some utility programs*

## 10.1 INTRODUCTION

The principal purpose of this chapter is to draw together all the preceding ideas in the past nine chapters into a series of example programs. As a bonus, these programs will be found to be particularly useful to you in your work with BASIC and with assembly language. In this sense they may be called *utility programs*. This is not to imply that programs in the rest of the book are not useful (for example Listing 8.6 is certainly of use if you have a printer). But the earlier programs were chosen to make specific pedagogical points: the programs here try to draw all the points already made together. If you purchase one or other of the tapes available with this book you will find some other utility programs. Tape 2 contains an assembly program to find and list the lines containing any section of code in a BASIC program (FINDCODE) and another to replace any section of code by any other section in a BASIC program (REPLACE). Tape 1 contains a 6502 disassembler (written in BASIC, but very fast): this will turn any section of machine code back into assembler mnemonics and make it easy for you to analyse all the BBC micro's firmware, as well as other people's machine code programs!

## 10.2 PROGRAM 1: RETRIEVE

The purpose of this program is to allow you to recover most or all of a program which has been corrupted in some way. It may be that you cannot get all the program from a damaged tape; or it may be that one of your machine code programs went a little wrong! In such cases, when you try to list the program you get the message 'Bad program'. In order to understand what happens here, we need to look a little into the workings of BBC BASIC.

Each line of a BASIC program, when it is stored in memory, begins with 4 bytes. The first is always &0D (ASCII for carriage return); the next is the high byte of the line number (which must not exceed &7F i.e. bit 7 must be zero); the next is the low byte of the line number; and the last is the number of bytes used in this particular line (including these four). This means that line numbers can range from 0 to 32767, and that the number of bytes allowed in a line, apart from the first four, is 251.

Now, apart from these first four, any line consists of two sorts of bytes: those with ASCII values between &20 and &7E, and those with values from &80 to &FF (the control codes from 0 to &1F and the delete code &7F will not be found in a BASIC program line). Those from &80 to &FF outside quotes represent keywords, and a list of these is given in your User Guide. Figure 10.1 shows how the line

```
    10PRINT"EXAMPLE":X=4:END
```

will go into memory (ignore the last byte for a moment). Notice how this method of storage makes the BASIC program entirely relocatable.

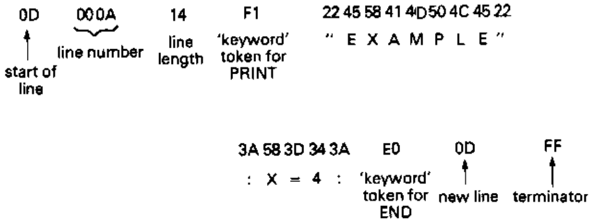


*Figure 10.1: How the program 10 PRINT "EXAMPLE": X = 4: END is stored in memory.*

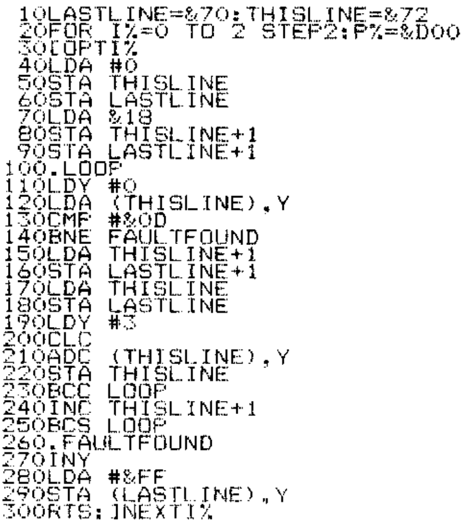
Now when you try to save or list a program, or when an error occurs (including ESCAPE) the interpreter first checks that the program is valid. It does this by seeing if the first byte is &0D: if not, it falls at the first hurdle. It now takes the next byte and checks its sign (BMI or BPL): if it is minus (i.e. bit 7 is 1) it knows it has reached the end of the program. This is because &FF is used as a terminator to the program, but any negative byte will do.

If it has not reached the end of the program, it adds the fourth byte (the line length) to the address of &0D and expects to find &0D at this new address - if it doesn't it outputs 'Bad program'. It repeats this process until either it finds the negative terminator or it outputs 'Bad program'.

Now our program RETRIEVE goes through a program making the same check, but when it comes to a point where the interpreter would output 'Bad program' it is much more friendly: it puts the terminator &FF at that point. In this way you have retrieved some (or all) of your program.

One last point. As Appendix 6 discusses, it is possible to start a program at any page by using the PAGE command. Location &18 contains the current page, and so we can use RETRIEVE on a program anywhere in memory, by first setting PAGE to the correct point.

LISTING 10.1



```
 10LASTLINE=&70:THISLINE=&72
 20FOR I%=0 TO 2 STEP2:P%=&D00
 30[OPTI%
 40LDA #0
 50STA THISLINE
 60STA LASTLINE
 70LDA &18
 80STA THISLINE+1
 90STA LASTLINE+1
100.LOOP
110LDY #0
120LDA (THISLINE),Y
130CMP #&0D
140BNE FAULTFOUND
150LDA THISLINE+1
160STA LASTLINE+1
170LDA THISLINE
180STA LASTLINE
190LDY #3
200CLC
210ADC (THISLINE),Y
220STA THISLINE
230BCC LOOP
240INC THISLINE+1
250BCS LOOP
260.FAULTFOUND
270INY
280LDA #&FF
290STA (LASTLINE),Y
300RTS:]NEXTI%
```

The program is in Listing 10.1, and the details are:

40-90 Initialise the low bytes of the pointers to the beginning of the last line analysed and to the present line under analysis, to zero; and the respective high bytes to the current page.

110 Initialise the byte pointer to the current line.

120-140 If the first byte of the line is not &0D we have found the fault - go to 270 to put in the terminator.

150-180 Otherwise put the address of the present line into the last line pointer.

190-250 And add to the present line pointer the length of the present line, storing this new address back into the present line pointer. Always branch back to 110.

270-300 Once the fault has been found, put the terminator &FF at the end of the last line and return. This will work even if the entire program can be recovered.

The program is relocatable, but is assembled into &D00. Save a machine code copy, and use it when you need to (which won't be very often, I hope).

## 10.3 PROGRAM 2: INTSORT

We saw in Listing 7.7 how to sort a set of up to 256 32-bit integers, and in Ex.7.3 how to generalise this to deal with more than 256 such integers. However, a major limitation to that program was that it could not sort an integer array created in a BASIC program e.g. by DIM ARRAY% (500), say. In this section we will see how to remedy this.

One feature of CALL we have not yet considered is its ability to pass parameters, or more precisely to pass addresses of parameters. For example, if we write CALL ANYTHING, Integer%, FPOINT, where ANYTHING is the start of the program,Integer% is an already declared integer variable and FPOINT an already declared decimal or floating-point variable (see Appendix 4), then a parameter block table will be set up starting at address &600. Figure 10.2 shows what it looks like.

| | | |
|---|---|---|
| &600 | 2 | Number of parameters passed |
| &601 | 56 | Address of first parameter is |
| &602 | 12 | &1256 |
| &603 | 4 | First parameters is an integer variable (4 bytes) |
| &604 | 82 | Address of second parameter is |
| &605 | 12 | &1282 |
| &606 | 5 | Second parameter is a floating-point variable (5 bytes) |

*Figure 10.2: The parameter block table in response to CALL ANYTHING, Integer %, FPOINT*

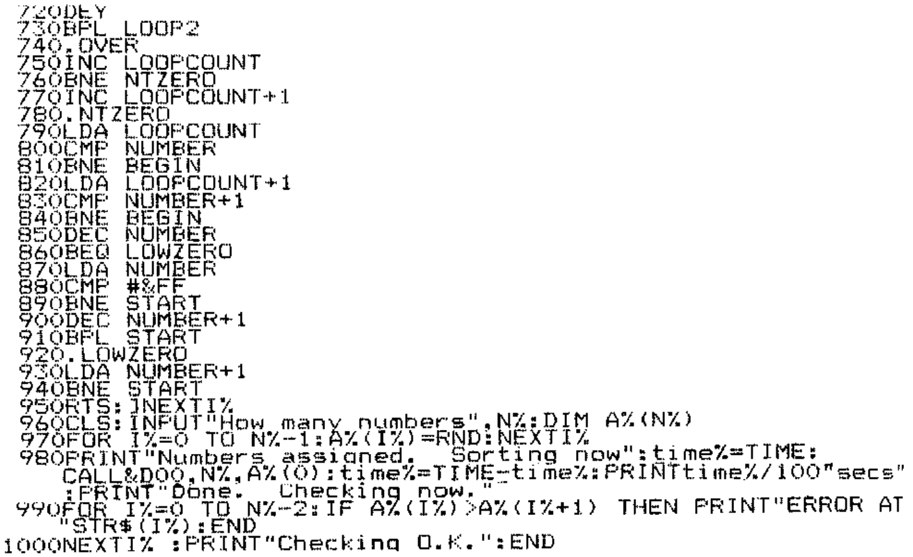
The first byte gives the number of parameters passed; and then each group of three following give the address of the relevant parameter together with its type: 4 is integer (4 bytes are used for an integer, so we expect &1256 to &1259 to contain our four byte Integer%); 5 is floating-point (FPOINT will occupy 5 bytes from &1282 to &1286). It is also possible to pass a single byte (e.g. ?BYTE) code 0 (1 would have been more sensible), and a string at a defined address (e.g. $TEXT) code 128, but both of these seem rather pointless since by definition we know where they are stored: in such cases, indirect indexed is the best course. Finally, as we shall see in the next section, we can pass string variables (e.g. String$).

Now it turns out that arrays are always arranged consecutively by the interpreter, so we can pass information about the whole of an integer array like one defined in DIM ARRAY% (500) by writing CALL ANYTHING, ARRAY% (0). From then on, we can add four to the address to get the next item in the array.

In our integer sorting program we need to pass one more item of information: the number of integers to be sorted. So we assume our program is called from BASIC with the statement CALL &D00, NUMBER%, ARRAY% (0) (the names NUMBER and ARRAY are arbitrary, but the % sign is essential).

LISTING 10.2

```
10TEMP=&70:FIRST=&71:SECOND=&73:NUMBER=&75:LOOPCOUNT=&77:
  OSWRCH=&FFEE
20FOR I%=0 TO 2 STEP 2:P%=&D00
30[OPTI%
40LDA &600
50CMP #2
60BNE MISTAKE
70LDA &603
80CMP #4
90BNE MISTAKE
100LDA &606
110CMP #4
120BEQ OK
130.MISTAKE
140LDA #ASC("?")
150JSR OSWRCH
160RTS
170.OK
180LDY #0
190LDA &601
200STA FIRST
210LDA &602
220STA FIRST+1
230LDA (FIRST),Y
240SEC
250SBC #1
260STA NUMBER
270INY
280LDA (FIRST),Y
290SBC #0
300STA NUMBER+1
310.START
320LDA &604
330STA SECOND
340LDA &605
350STA SECOND+1
360LDA #0
370STA LOOPCOUNT
380STA LOOPCOUNT+1
390.BEGIN
400LDY #0
410LDA SECOND+1
420STA FIRST+1
430LDA SECOND
440STA FIRST
450CLC
460ADC #4
470STA SECOND
480BCC NOCARRY
490INC SECOND+1
500.NOCARRY
510LDX #4
520SEC
530.LOOP1
540LDA (SECOND),Y
550SBC (FIRST),Y
560INY
570DEX
580BNE LOOP1
590BVC NOOVFLOW
600EOR #&80
610.NOOVFLOW
620EOR #0
630BPL OVER
640DEY
650.LOOP2
660LDA (FIRST),Y
670STA TEMP
680LDA (SECOND),Y
690STA (FIRST),Y
700LDA TEMP
710STA (SECOND),Y
```



```
720DEY
730BPL LOOP2
740.OVER
750INC LOOPCOUNT
760BNE NTZERO
770INC LOOPCOUNT+1
780.NTZERO
790LDA LOOPCOUNT
800CMP NUMBER
810BNE BEGIN
820LDA LOOPCOUNT+1
830CMP NUMBER+1
840BNE BEGIN
850DEC NUMBER
860BEQ LOWZERO
870LDA NUMBER
880CMP #&FF
890BNE START
900DEC NUMBER+1
910BPL START
920.LOWZERO
930LDA NUMBER+1
940BNE START
950RTS:]NEXTI%
960CLS:INPUT"How many numbers",N%:DIM A%(N%)
970FOR I%=0 TO N%-1:A%(I%)=RND:NEXTI%
980PRINT"Numbers assigned.  Sorting now":time%=TIME:
   CALL&D00,N%,A%(0):time%=TIME-time%:PRINTtime%/100"secs"
   :PRINT"Done.  Checking now."
990FOR I%=0 TO N%-2:IF A%(I%)>A%(I%+1) THEN PRINT"ERROR AT
   "STR$(I%):END
1000NEXTI% :PRINT"Checking O.K.":END
```

The program is in Listing 10.2 and the details are:

40-60 If there are not two parameters, something is wrong.

70-120 And if they are not both integers, something is wrong.

140-160 In these cases, output a query sign and return.

180-300 Store the address of NUMBER% in a temporary location (FIRST will be used for other things later). Subtract one from the contents of that address and store in NUMBER. Store the byte from the next address in NUMBER + 1 (adjusting it if any borrow had occurred on the previous subtraction).

320-350 Store the address of the first integer (ARRAY%(0)) temporarily in SECOND.

360-380 Initialise the integer counter.

400-730 Identical to 120-450 of Listing 7.7. Notice how in 670 and 700 we do not use PHA and PLA since they cost one cycle in a very often used loop.

750-760 If the lower byte of the integer count is not zero, go straight to 790 where we can compare it to the lower byte of NUMBER.

770 If it is zero, increment the higher byte of the count first.

790-810 If the lower bytes don't agree, we can't be finished with the present cycle through the integers. Go back to 400.

820-840 Even if the lower bytes agree, equality cannot occur unless the higher bytes agree too. Go back to 400.

850-910 At the end of the present cycle, decrement the lower byte of NUMBER by 1. If it is not zero, examine whether it has gone through zero: if so, decrement the high byte of NUMBER by 1. Return to 320 in all cases, unless the high byte of NUMBER has passed through zero (which will never happen - see 930-950).

930-950 This checks if the high byte of NUMBER has got to zero as well as the low byte of NUMBER, since entry to 930 is from 860 only. In this case, (NUMBER) equals 0 and we have finished; otherwise return to 320.

Notice how much more code is necessary when the number of integers can exceed 255: 6 lines of code for fewer than 256 integers, 18 lines for more than 256 integers. The principal problem is that we need to allow the case where the low byte of NUMBER equals zero in every cycle, except the one where the high byte is also zero (and for fewer than 256 integers this will be the only such case where the lower byte is zero). Hence, we need to check if the high byte of NUMBER is zero every time the low byte reaches zero.

This is not the fastest machine code sort possible, but it is simple and still pretty quick. To sort 2000 integers in BASIC using the bubble sort would take all day: see how long it takes in machine code - about 190 seconds. (If you have a model A, to give yourself room insert line 955 STOP; run the program; delete lines 10 to 955 and run it again.) In practice, though, you are not likely to want to sort more than about 500 integers, and this takes only 12 seconds. (The reason for the sixteen-fold increase from 500 to 2000 is that the number of comparisons with 500 is ½ × 500 × 501 i.e. 125,250 whereas with 2000 it is ½ × 2000 × 2001 i.e. 2,001,000, almost 16 times as great. Incidentally, this means that each comparison occupies about 0.0001 second or about 200 machine cycles.)

## 10.4 PROGRAM 3: STRINGSORT

Let us now do the same thing for Listing 7.8 as we did for Listing 7.7. There is a slight complication when we pass a string variable via CALL. The address we get is not of the string, but of another block of information called a String Information Block. This consists of four bytes: the first two give the starting address of the string and the last one the length of the string. This is essential because a string variable does not end in &0D unlike $TEXT.

The third byte is of no use to us here, but it is quite interesting: it denotes the maximum length of string possible without reallocation of space. When a string variable is first defined in

BBC BASIC it is given a bit more space than it actually needs: this amount of space is stored in the third variable. The reason for doing this is to cut down on what are called garbage collection problems. When we define A$ to be "ABC" and then later define it to be A$ + A$ + A$ + A$ + A$, this new string will have to be stored somewhere else in memory: the old "ABC" remains where it was. What will happen here is that the string information block will have its contents changed to point to the new string (but the block itself will not need to be moved). If we do this a lot, we will run out of memory, but a lot of memory will be old discarded strings - in other words, garbage. So all this garbage will need to be collected up and thrown away; and this is a slow process, even in machine code. By careful programming, however, we can assign enough space to our original string to circumvent this. As long as our new allocation does not exceed byte three, we will not have to use up any more memory to store our new string: we just overwrite the old one. So, if we know a string is unlikely to be much more than about 25 characters, define it initially as A$ = STRING$(25, "*").

Back now to our immediate task. As far as we are concerned then, a string array will be a continuous set of bytes in memory, each element of the array corresponding to four of those bytes (the String Information Block). The strings themselves can be anywhere else in memory: the first two bytes in the String Information Block point to the starting point of the string. It follows, that the first half of Listing 7.8 has already been done for us by the interpreter: the list of pointers already exists, we do not have to create it. However, we must remember that when we swap pointers there are four bytes to swap: the address plus the two string length bytes. We shall assume that there are no null strings in the set of strings we shall be sorting. A null string is denoted by zeros in the string information block.

LISTING 10.3

```
 10LOOPCOUNTH=&70:FIRST=&71:SECOND=&73:TEMP=&75:ADDRESS=&76:
   NUMBER=&78:STORE1= &7A:STORE2=&7C:LGTH1=&7E:LGTH2=&7F:
   OSWRCH=&FFEE
 20FOR I%=0 TO 2 STEP 2:P%=&D00
 30[OPTI%
 40LDA &600
 50CMP #2
 60BNE MISTAKE
 70LDA &603
 80CMP #4
 90BNE MISTAKE
100LDA &606
110CMP #&81
120BEQ OK
130.MISTAKE
140LDA #ASC("?")
150JSR OSWRCH
160RTS
170.OK
180LDA &601
190STA FIRST
200LDA &602
210STA FIRST+1
220LDY #0
230LDA (FIRST),Y
240SEC
250SBC #1
260STA NUMBER
270INY
280LDA (FIRST),Y
```

```
290SBC #0
300STA NUMBER+1
310.LOOP3
320LDA &604
330STA STORE2
340LDA &605
350STA STORE2+1
360LDX #0
370STX LOOPCOUNTH
380.LOOP4
390LDY #0
400LDA STORE2+1
410STA STORE1+1
420LDA STORE2
430STA STORE1
440CLC
450ADC #4
460STA STORE2
470BCC NOCARRY3
480INC STORE2+1
490.NOCARRY3
500LDA (STORE1),Y
510STA FIRST
520LDA (STORE2),Y
530STA SECOND
540INY
550LDA (STORE1),Y
560STA FIRST+1
570LDA (STORE2),Y
580STA SECOND+1
590LDY #3
600LDA (STORE1),Y
610STA LGTH1
620LDA (STORE2),Y
630STA LGTH2
640LDY #0
650.LOOP5
660LDA (FIRST),Y
670CMP (SECOND),Y
680BCC NEWRECORD
690BNE SWAP
700INY
710CPY LGTH1
720BEQ NEWRECORD
730CPY LGTH2
740BEQ SWAP
750BNE LOOP5
760.PIVOT1
770BPL LOOP3
780.PIVOT2
790BNE LOOP3
800.SWAP
810 LDY #3
820.LOOP6
830LDA (STORE1),Y
840STA TEMP
850LDA (STORE2),Y
860STA (STORE1),Y
870LDA TEMP
880STA (STORE2),Y
890DEY
900BPL LOOP6
910.NEWRECORD
920INX
930BNE NTZERO
940INC LOOPCOUNTH
950.NTZERO
960CPX NUMBER
970 BNE LOOP4
980LDA LOOPCOUNTH
990CMP NUMBER+1
1000BNE LOOP4
1010DEC NUMBER
1020BEQ LOWZERO
1030LDA NUMBER
1040CMP #&FF
1050BNE LOOP3
1060DEC NUMBER+1
1070BPL PIVOT1
1080.LOWZERO
1090LDA NUMBER+1
1100BNE PIVOT2
1110RTS
```

```
1120.TEXT:]NEXTI%
1130CLS:INPUT'"How many records",N%
1140DIM A$(N%-1)
1150PRINT'"Setting up strings now"
1160 S=0:FOR I%=0 TO N%-1:R=RND(10):A$="":FOR J%= 0 TO R-1:
   A$=A$+CHR$(RND(26)+64):NEXTJ%:S=S+R:A$(I%)=A$:NEXTI%
1170PRINT"Sorting now.":time%=TIME:CALL&D00,N%,A$(0):
   time%=TIME-time%:PRINTtime %/100"secs":PRINT"Checking."
1180FOR I%=0 TO N%-2:IF A$(I%)>A$(I%+1) THEN PRINT"ERROR AT
   "STR$(I%):END
1190NEXT:PRINT"O.K.":END
```

The program is in Listing 10.3 and the details are:

40-160 Check that the parameters are of the right type. The code for a string variable is &81. If there are any mistakes output a query and return.

180-300 As Listing 10.2.

320-350 As 400-430 of Listing 7.8 except that we do not need STORE: the information is in &604 and &605.

360-370 Low byte of string count is in X which is not otherwise required in the program.

390-580 As 460-650 of Listing 7.8 except that we need to add 4 not 2 to get the next pointer.

590-630 Store the lengths of the respective strings being compared in LGTH1 and LGTH2.

640-700 As 660-720 of Listing 7.8.

710-720 If we have reached the end of the first string and there is still equality, no swap is needed.

730-740 By contrast, if the end of the second string is reached a swap is required.

750 If we get to here, the branch will always occur.

760-790 See 920-1100.

810-900 As 750-840 of Listing 7.8 except that we swap four bytes not two.

920-1100 As 750-940 of Listing 10.2, except that the low byte of the count is in X, not in LOOPCOUNT, and LOOPCOUNT+1 is replaced by LOOPCOUNTH. Using X is quicker, and allows a saving at 960 (compared to 790 and 800 of Listing 10.2).

Lines 1070 and 1100 are necessary in order to create a relocatable program. The jump is just too far for a straight branch, and so we use pivots at 760-790. Listing 10.2 was slightly easier to make relocatable than this one, but the cost is still small, and is certainly worth it for a program of this sort (no pun intended!).

Try it for 1000 strings (with a Model A, with luck and the same method as in the last section, this ought to just fit: if not change 10 to 8 in 1160). Again, not the fastest sort possible, but still very reasonable compared to BASIC (BASIC takes long enough to set up the test strings!). You should find the sort takes about 48 seconds. (Again there are ½ × 1000 × 1001 = 500,500 comparisons, so each comparison takes about 0.0001 second, giving 200 machine cycles per comparison on average. This is just the same as with Listing 10.2, and will not alter much with much longer strings, as long as they are not too close to each other. For example, re-running the program with 32 in 1160 for 250 strings gives a sorting time of about 3 seconds, one sixteenth of the 1000 string sort as we would expect. And 4 in 1160 with 1500 strings gives about 108 seconds, 2.25 times as much as the 1000 string sort, again as we would expect.)

## 10.5 PROGRAM 4: REMSPACE

When you type a BASIC program into your computer it is useful to leave a liberal number of spaces to aid legibility. For example, IF X > 3 THEN PRINT "MORE" is much easier to read than IFX>3THENPRINT"MORE". Moreover, in a case like IF X > Y THEN 100, the space between Y and THEN is essential otherwise BASIC will look for a variable YTHEN.

Again, to aid understanding it is useful to put in the occasional REM statement either in a full line, such as

```
      10     REM This is a full line
```

or a part line like

```
      50     X = 0 : Y = 0 : REM This is a part line
```

The problem with both these strategies is that they take up memory space; and in a microcomputer like the BBC's, space can be crucial in high-resolution modes.

The purpose of our next program is to solve this problem. All spaces and all REM statements will be extracted from any program (of course, spaces occurring between quotes will be left). You can therefore keep two copies of a program: one for running and one for documentation purposes. (If you have REPLACE, available on the second tape, you can reduce the program still further by replacing all variables by single letter variables.)

Once a statement like IF X > Y THEN 100 is translated into BASIC, using tokens for IF and THEN, it is perfectly all right to remove all the spaces, so no precautions are needed here. This assumes that you don't use lines like .LABEL DEY, for if you do, when the space is removed the computer will treat it as .LABELDEY and the DEY instruction will be lost into the label. If you must use this, write .LABEL: DEY. Again DEFPROCYN Y% = -1 will lead to difficulties, since the procedure will take on the new name YNY% and you will get a 'no FN/PROC' error. There is no need to use this formulation - use DEFPROCYN: Y% = -1; it is much easier to read anyway.

When full line REMs are removed any GOTO etc will point to the wrong place if it hitherto pointed to the REM. My own preference is never to point to a REM statement in this way; but if your own preference is different, you will have to change all the line numbers yourself, (or write a utility to do it for you!).

The program is shown in Listing 10.4. It isn't relocatable because to do without the subroutine MVEMEM would be a substantial hardship. However, if you want it to be relocatable, you can put the subroutine permanently at location &75 onwards (it takes up 8 bytes) and define MVEMEM to be &75 in line 10. Delete lines 820 to 870, and insert the following 16 lines, after writing 21 [OPT I%:

```
22      LDA #&B1
23      STA &75
24      LDA #&72
25      STA &76
26      LDA #&91
27      STA &77
28      LDA #&70
29      STA &78
30      LDA #&C8
31      STA &79
32      LDA #&C9
33      STA &7A
34      LDA #&0D
35      STA &7B
36      LDA #&60
37      STA &7C
```

Now everytime the program is run, it will write the subroutine MVEMEM into &75 to &7C, no matter where you put the program. My own preference is not to go to these extremes to create relocatable code, but I offer this solution to those of you who feel more strongly about it.

In the program, two main variables are used: NEWLINE is the address of the first byte of the current line of the 'new' program (i.e. the program with spaces and REMs removed); OLDLINE is the address of the current byte of the 'old' program less the pointer to the current byte of the 'new' program (which is stored in Y) i.e. NEWLINE + Y = current byte of 'new' program; OLDLINE + Y = current byte of 'old' program.

The method used is to transfer the program byte by byte, not transferring spaces or REM statements. In a sense, the program is transferred back into its own memory space, but it will take up less of that space at the end. OLDLINE is incremented by one every time that a space is met or (FLAGREM)$_7$ is set. (FLAGREM)$_7$ is set to one if a REM has been met. This will cause all further bytes of the 'old' line to be passed over: OLDLINE is incremented but not Y, the 'new' line byte pointer.

LISTING 10.4

```
   10NEWLINE=&70:OLDLINE=&72:FLAGREM=&74
   20FOR I%=0 TO 2 STEP2:P%=&D00
   30[OPTI%
   40LDA #0
   50STA NEWLINE
   60STA OLDLINE
   70LDA &18
   80STA NEWLINE+1
   90STA OLDLINE+1
  100.LOOP1
  110LDA #0
  120STA FLAGREM
  130LDY #1
  140JSR MVEMEM
  150CMP #&FF
  160BEQ FINISH
  170JSR MVEMEM
  180JSR MVEMEM
  190.LOOP2
  200LDA (OLDLINE),Y
  210BIT FLAGREM
  220BPL NTSET
  230CMP #&0D
  240BNE SPACE
  250JSR MVEMEM
  260BEQ ENDLINE
  270.NTSET
  280CMP #&20
  290BEQ SPACE
  300CMP #&F4
  310BNE NTREM
  320DEY
  330LDA #&80
  340STA FLAGREM
  350BNE SPACE
  360.NTREM
  370JSR MVEMEM
  380BEQ ENDLINE
  390CMP #&22
  400BEQ QUOTES
  410BNE LOOP2
  420.SPACE
  430INC OLDLINE
  440BNE LOOP2
  450INC OLDLINE+1
  460BNE LOOP2
  470.ENDLINE
  480DEY
  490TYA
  500PHA
  510CPY #3
  520BEQ NOCARRY
  530LDY #3
  540STA (NEWLINE),Y
  550CLC
  560ADC NEWLINE
  570STA NEWLINE
  580BCC NOCARRY
  590INC NEWLINE+1
  600.NOCARRY
  610PLA
  620CLC
  630ADC OLDLINE
  640STA OLDLINE
  650BCC LOOP1
  660INC OLDLINE+1
  670BNE LOOP1
  680.QUOTES
  690JSR MVEMEM
  700BEQ ENDLINE
  710CMP #&22
```

```
720BNE QUOTES
730BEQ LOOP2
740.FINISH
750LDA NEWLINE
760CLC
770ADC #2
780STA &12
790LDA NEWLINE+1
800ADC #0
810STA &13
820RTS
830.MVEMEM
840LDA (OLDLINE),Y
850STA (NEWLINE);Y
860INY
870CMP #&0D
880RTS:]NEXT
```

The details are:

40-90 Put address of beginning of program to be 'compacted', into the pointers.

110-120 Set the 'REM-found' flag to zero initially, at the beginning of each line of the program to be compacted.

130-140 Move the high byte of the address from the 'old' to the 'new' line.

150-160 If this high byte is &FF, the end of the program is reached.

170-180 Otherwise, move the low byte of the address and the line length byte, from the 'old' to 'new' lines.

200 Put the current byte of the 'old' line of the program into the accumulator.

210-220 If FLAGREM not set (i.e. bit 7 = 0) branch to 280.

230-240 Otherwise check if end of line, and if not, branch to 430.

250-260 If end of line is reached, move &0D into the 'new' line and always branch to 480.

280-290 Check if space, and if so branch to 430.

300-310 Check if REM token, and if not skip over to 370.

320 Otherwise reduce 'new' line byte pointer by one, so that the colon before the REM is written over by &0D in lines 250-260. If this is a full line REM, this decrement is irrelevant, but will be performed anyway.

330-360 Set bit 7 of FLAGREM and always branch to 430.

370-380 Move a byte from 'old' to 'new' line, and branch to 480 if end of line is reached.

390-410 If quote met, branch to 690; otherwise go back to 200.

430-460 Increment the pointer to the 'old' line by one and return to 200.

470 Routine when end of line reached.

480-500 Reduce the 'new' line byte pointer by one (it was incremented unnecessarily in 860) and transfer to the stack.

510-520 If this pointer is 3, a full line REM was met, so skip over update of NEWLINE by going straight to 610.

530-540 Store this pointer as new line-length byte in the 'new' line.

550-590 Update NEWLINE to beginning of next 'new' line.

610-670 Update OLDLINE to beginning of next 'old' line.

680 Routine entered when quote encountered.

690-730 As long as another quote is not met, keep moving bytes from the 'old' to 'new' line, checking for end of line at each stage. Return to 200 if another quote met.

750-820 Reset TOP location (&12 and &13) to point to top of the 'new' program, and return to call.

840-870 Move a byte from the 'old' to the 'new' line, increment the 'new' line byte pointer by 1, and set Z to 1 if end of line reached.

## 10.6 PROGRAM 5: MEMORYHUNT

Our next program searches any section of memory for any set of bytes and displays the addresses (if any) of their location in that section of memory.

It is particularly useful if you want to analyse the workings of the BBC micro's firmware. Used in conjunction with a disassembler (there is one on the first tape available with this book) it offers a powerful tool for understanding how the interpreter and operating system work. It differs in its structure a little from the rest of the programs in this chapter in that it is a mixture of BASIC and assembly language. We use BASIC to perform all the inputs, error checking and initialisation; and we use assembly language to very speedily search through memory.

LISTING 10.5

```
10GOSUB 240:*KEY9"GOTO 100|M"
20VDU12:PRINT''"Do you want to search for a string."'
  "some machine code or some assembly"'"code (S/M/A)?"
  30A$=GET$:IFA$="A"THENPRINT''"Input the assembly code
  in line 130"'"of this program. When you have"'"finished
  press key f9":END
40IFA$="S"THEN80ELSEIFA$<>"M"THEN30
50INPUT"How many bytes",BYTES%:PRINT"Input them one by
  one in order."'"Hex is assumed so do not prefix with &."'
60FORI%=1TOBYTES%:PRINT"BYTE";STR$(I%);:INPUTA$:?(&6F+I%)
  =EVAL("&"+A$):NEXT
70GOTO90
80INPUT"What is the string",A$:FORI%=1TOLEN(A$):?(&6F+I%)
  =ASC(MID$(A$,I%,1)): NEXT
```

```
 90BYTENUMBER=I%-1:GOTO150
100GOSUB240
110 ON ERROR PRINT"Assembly error: correct line"'"130
    and press f9":END
120FORI%=0TO2STEP2:P%=&70:[OPTI%
130JMP (&206):REM MNEMONICS SEARCHED PUT HERE
140]NEXT:BYTENUMBER=P%-&70
150ON ERRORPRINT'"Hex is assumed so do not prefix with &."
160PRINT'"The default starting and finishing"'"addresses
    of memory to be searched"'"are &8000 to &FFFF."'"Is this
    alright (Y/N)?"
170A$=GET$:IFA$="Y"THENA$="8000":B$="FFFF" ELSEIFA$=
    "N"THENPRINT''"What are the starting and finishing":
    INPUT"addresses of memory to be searched",A$,B$ ELSEGO
    TO170
180 START%=EVAL("&"+A$):FINISH%=EVAL("&"+B$):?&8E=START
    %MOD256:?&8F=START%DIV2 56:ON ERROR OFF
190PRINT'"Print out display (Y/N)?";:A$=GET$:IF A$="Y"
    THEND%=0 ELSE IF A$="N"THEN D%=1 ELSE 190
200 SUM%=FINISH%-START%+1:IFSUM%=&10000THENSUM%=&FFFF
210 ?&8C=SUM%MOD256:?&8D=SUM%DIV256:?&89=BYTENUMBER:VDU12,
    D%+2:CALL START:IFD% =0 THENVDU1,13,3
220PRINT'"E  To end        R  To rerun program"
230A$=GET$:IFA$="R"THEN20ELSEIFA$<>"E"THEN230ELSEEND
240VDU12:DIM START 250:BYTECOUNT=&89:MATCHBASE=&8A:
    MEMTOTAL=&8C:SEARCHBASE=&8E :OSNEWL=&FFE7:OSWRCH=&FFEE
250FORI%=0TO2STEP2:P%=START
260[OPTI%
270JSR OSNEWL
280LDY #0
290LDA MEMTOTAL+1
300BEQ NOHIGH
310.LOOP1
320JSR COMPARE
330INY
340BNE LOOP1
350INC SEARCHBASE+1
360DEC MEMTOTAL+1
370BNE LOOP1
380.NOHIGH
390LDA MEMTOTAL
400BEQ DONE
410.LOOP2
420JSR COMPARE
430INY
440DEC MEMTOTAL
450BNE LOOP2
460.DONE
470RTS
480.COMPARE
490LDA (SEARCHBASE),Y
500CMP &70
510BEQ MATCH
520RTS
530.MATCH
540LDX #0
550TYA
560PHA
570LDY #0
580CLC
590ADC SEARCHBASE
600STA MATCHBASE
610LDA SEARCHBASE+1
620ADC #0
630STA MATCHBASE+1
640.LOOP3
650INX
660CPX BYTECOUNT
670BEQ FOUND
680INY
690LDA (MATCHBASE),Y
700CMP &70,X
710BEQ LOOP3
720PLA
730TAY
740RTS
750.FOUND
760LDA MATCHBASE+1
770JSR HEXASC
780LDA MATCHBASE
790JSR HEXASC
800LDA #&20
```

```
 810JSR OSWRCH
 820PLA
 830TAY
 840RTS
 850.HEXASC
 860PHA
 870LSR A
 880LSR A
 890LSR A
 900LSR A
 910JSR CONVT
 920PLA
 930AND #&0F
 940JSR CONVT
 950RTS
 960.CONVT
 970SED
 980CLC
 990ADC #&90
1000ADC #&40
1010CLD
1020JSR OSWRCH
1030RTS:]NEXT
1040RETURN
```

The program is in Listing 10.5. The BASIC part is from 10 to 230. The idea is that we can input the bytes simply as a series of bytes (lines 50 to 60) or as a string (line 80) or as assembler mnemonics (line 30, and 100-140). In this latter case, we must leave the program to input the mnemonic code in line 130, and then re-enter at 100 using the programmable $f_9$ key. The machine code routine will have to be reassembled (GOSUB 240) in this case to deal with destruction of dynamic memory caused by the re-writing of 130. In the case of bytes and strings no re-assembly is necessary and we may continue to re-run the program (from line 220) without doing a GOSUB 240. The rest of the BASIC program sets up all the values required by the machine code routine, deals with the various input requirements and then calls the machine code.

Lines 270-1030 contain the assembly program. The program is written in such a way that if we are searching for the string ABA then it will be found twice in ABABA. This is the best strategy for a memory-searching problem of this type. The code to be searched for is put in &70 onwards and BYTENUMBER contains the number of bytes. There are three variables used for the searching process: MEMTOTAL contains the total number of bytes to be searched. SEARCHBASE contains the base address of the memory to be searched. It is used in conjunction with the index register Y, and, employing indirect indexed addressing the high byte is incremented every 256 bytes. MATCHBASE contains the address of the current section of memory being compared to the code searched, where the first byte of this section is the same as the first byte of the code to be searched.

The details are:

270 Print a new line to guard against overscanning on ordinary TVs (which tends to obscure the top line).

280 Set pointer to current byte at zero.

290-300 If fewer than 256 bytes are to be searched, jump to 390.

320 See if first byte of code searched matches current byte (at 490).

330-340 Loop for 256 bytes.

350-370 Increment base pointer to memory searched, decrement byte total and return to loop if high byte of total still not zero.

390-400 If low byte zero, finish.

420 Again compare first byte of code to current byte (at 490).

430-470 Increment pointer, decrement low byte of total and continue until zero, when return.

490-520 Compare current byte to first byte of code looked for. If it matches, go to 540; otherwise, return to the searching of memory.

540 Set pointer to code looked for at zero.

550-560 Save Y on the stack.

570 Treat Y as a pointer relative to current match position and set to zero.

580-630 Store address of current match position in MATCH-BASE.

650 Increment pointer to code looked for.

660-670 If all bytes have been matched, jump to address print-out (at 760).

680-710 Otherwise load next byte of memory, compare with next byte of code looked for, and if equal loop back to 650.

720-740 If not equal, match unsuccessful. Restore value of Y before entry to MATCH routine, and return to searching of memory.

750 Routine to print out the address of the first byte where match achieved.

760-810 Display high and low bytes of beginning of match in memory, followed by a space.

820-840 Restore the old value of Y prior to entry to MATCH, and return to searching of memory. This ensures that if we are looking for, say, ABA then it is found twice in ABABA. If Y were set to the position immediately following the end of the match, ABA would only be located once, since the search would resume at the beginning of the second BA, instead of at the beginning of the first BA, as here.

850 Routine to display byte on the screen.

860-910 Save byte on the stack, move top nybble to bottom nybble position, and go to conversion routine (at 970).

920-950 Retrieve byte from the stack, set top nybble to zero, go to conversion routine and return.

960 Converts a single hex digit to ASCII and prints it on the screen.

970-990 Using BCD, add 90. If digit is between 0 and 9, no carry is generated. If between A and F, a carry is generated into the top nybble and thence to the carry flag (e.g. 0B → $\boxed{1}$01).

1000 Add 40 with carry BCD. For 90 to 99 this gives 30 to 39. For $\boxed{1}$00 to $\boxed{1}$05, this gives 41 to 46.

1010-1030 Clear BCD flag, output the ASCII digit and return.

## 10.7 PROGRAM 6: MC-MONITOR

When working in assembly language or machine code it is very important to be able to monitor the progress of a program, especially if it does not work in quite the way you wanted it to. What is required is a program which will at least:

(i) display all the registers - SP, X, Y, A and each bit of the flag (status) register;

(ii) display the contents of the stack;

(iii) display any memory location, but especially the zero page locations from &70 to &8F;

(iv) allow you to alter the X, Y, A or P registers or any memory location; and

(v) allow you to go to any subroutine and return to the display in (i)-(iii) above.

In addition, if it is to be an effective debugging tool, it must allow you to jump into it from the program you are trying to debug, and to return to the same point in the program when you are ready.

This is the purpose of our last program, shown in Listing 10.6. It is a very much bigger program than any other in this book, occupying about 800 bytes, and it is assembled into the top of memory using the HIMEM method (see 2.5). If you do not already have the program on tape, type it in and save it on tape. We shall look first at what the program does; and then at how it works.

LISTING 10.6

```
10HIMEM=HIMEM-&330
20OSWRCH=&FFEE:OSRDCH=&FFE0:OSNEWL=&FFE7:OSASCI=&FFE3:
  OSBYTE=&FFF4
30S%=HIMEM:FORI%=0 TO 2 STEP 2
40P%=S%
50[OPTI%
60.START
70PHP
80PHA
90TYA
```

```
100PHA
110TXA
120PHA
130CLD
140CLI
150LDA#12
160JSR OSWRCH
170LDX #0
180JSR OUTPUT
190TSX
200TXA
210CLC
220ADC #6
230PHA
240TAX
250INX
260BEQ NOSTACK
270.STACK
280LDA &100,X
290JSR HEXASC
300LDA#&20
310JSR OSWRCH
320JSR OSWRCH
330JSR OSWRCH
340INX
350BNE STACK
360.NOSTACK
370JSR OSNEWL
380LDX #(TEXT2-TEXT)
390JSR OUTPUT
400LDA #&20
410JSR OSWRCH
420JSR OSWRCH
430TSX
440LDY #4
450.REGST
460INX
470LDA &100,X
480JSR HEXASC
490LDA #&20
500JSR OSWRCH
510DEY
520BNE REGST
530INX
540LDA &100,X
550LDX #8
560.STATUS
570ASL A
580PHA
590LDA #&30
600BCC ZERO
610LDA #&31
620.ZERO
630JSR OSWRCH
640PLA
650DEX
660BNE STATUS
670JSR OSNEWL
680LDX #(TEXT3-TEXT)
690JSR OUTPUT
700LDA #0
710STA MEMLOC2+1
720STA MEMLOC4+2
730LDA #&70
740STA MEMLOC3+1
750STA MEMLOC4+1
760JSR MEMOUT
770.MENU
780LDX #(TEXT4-TEXT)
790JSR OUTPUT
800LDX #(TEXT5-TEXT)
810JSR OUTPUT
820.CHAR
830JSR INCHAR
840CMP #2
850BEQ CONTROL
860CMP #3
870BEQ CONTROL
880CMP #14
890BEQ CONTROL
900CMP #15
910BNE NTCONTROL
920.CONTROL
```

```
 930JSR OSWRCH
 940.NTCONTROL
 950CMP #ASC("E")
 960BNE ENOT
 970LDX #&FD
 980TXS
 990RTS
1000.ENOT
1010CMP #ASC("C")
1020BNE CNOT
1030PLA
1040PLA
1050TAX
1060PLA
1070TAY
1080PLA
1090PLP
1100RTS
1110.CNOT
1120CMP #ASC("M")
1130BEQ MTYPE
1140CMP #ASC("G")
1150BEQ GTYPE
1160CMP #ASC("R")
1170BEQ RTYPE
1180CMP #ASC("A")
1190BNE CHAR
1200JSR MORA
1210STX MEMLOC8+1
1220JSR BYTE
1230STX MEMLOC8+2
1240JSR INBYTE
1250.MEMLOC8
1260STX &FFFF Dummy
1270JMP NOSTACK
1280.MTYPE
1290JSR MORA
1300STX MEMLOC4+1
1310STX MEMLOC3+1
1320JSR BYTE
1330STX MEMLOC2+1
1340STX MEMLOC4+2
1350JSR MEMOUT
1360JMP MENU
1370.GTYPE
1380JSR MORA
1390STX MEMLOC9+1
1400JSR BYTE
1410STX MEMLOC9+2
1420PLA
1430PLA
1440TAX
1450PLA
1460TAY
1470PLA
1480PLP
1490.MEMLOC9
1500JSR &FFFF Dummy
1510JMP START
1520.RTYPE
1530LDX #(TEXTB-TEXT)
1540JSR OUTPUT
1550.GTREG
1560JSR INCHAR
1570CMP #&0D
1580BEQ JUMP
1590 TSX
1600INX
1610INX
1620CMP #ASC("X")
1630BEQ REGGOT
1640 INX
1650CMP #ASC("Y")
1660BEQ REGGOT
1670 INX
1680CMP #ASC("A")
1690BEQ REGGOT
1700INX
1710CMP #ASC("P")
1720BNE GTREG
1730.REGGOT
1740JSR OSWRCH
```

```
1750TXA
1760PHA
1770JSR INBYTE
1780TXA
1790TAY
1800PLA
1810TAX
1820TYA
1830STA &100,X
1840.JUMP
1850JMP NOSTACK
1860.MORA
1870PLA
1880 STA RETADRL3+1
1890PLA
1900STA RETADRH3+1
1910.NTENOUGH2
1920LDX #(TEXT6-TEXT)
1930JSR OUTPUT
1940JSR HEXINPUT
1950CPY #5
1960BEQ OK2
1970CPY #1
1980BNE PULL2
1990PLA
2000JMP NOSTACK
2010.PULL2
2020PLA
2030DEY
2040BNE PULL2
2050JSR MISTAKE
2060JMP NTENOUGH2
2070.OK2
2080PLA
2090JSR OSNEWL
2100JSR BYTE
2110.RETADRH3
2120LDA #0 Dummy
2130PHA
2140.RETADRL3
2150LDA #0 Dummy
2160PHA
2170RTS
2180.HEXINPUT
2190PLA
2200STA RETADRL2+1
2210PLA
2220STA RETADRH2+1
2230.AGAIN
2240LDY #0
2250.DIGIT
2260JSR INCHAR
2270CMP #&7F
2280BEQ OVERCHECK
2290CMP #&0D
2300BEQ OVERCHECK
2310CMP #&30
2320BCC DIGIT
2330CMP #&47
2340BCS DIGIT
2350CMP #&3A
2360BCC OVERCHECK
2370CMP #&41
2380BCC DIGIT
2390.OVERCHECK
2400JSR OSASCI
2410CMP #&7F
2420BNE NTDEL
2430DEY
2440BMI AGAIN
2450PLA
2460JMP DIGIT
2470.NTDEL
2480PHA
2490INY
2500CMP #&0D
2510BNE DIGIT
2520.RETADRH2
2530LDA #0 Dummy
2540PHA
2550.RETADRL2
2560LDA #0 Dummy
```

```
2570PHA
2580RTS
2590.OUTPUT
2600JSR OSNEWL
2610.MEMLOC1
2620LDA TEXT,X
2630JSR OSASCI
2640INX
2650CMP #&0D
2660BNE MEMLOC1
2670RTS
2680.MEMOUT
2690LDY #4
2700.MEMLOC2
2710LDA #0 Dummy
2720JSR HEXASC
2730.MEMLOC3
2740LDA #0 Dummy
2750JSR HEXASC
2760LDA #&3A
2770JSR OSWRCH
2780LDA #&20
2790JSR OSWRCH
2800LDX #0
2810.MEMLOC4
2820LDA &FFFF,X  Dummy
2830JSR HEXASC
2840LDA #&20
2850JSR OSWRCH
2860INX
2870CPX #8
2880BCC MEMLOC4
2890JSR OSNEWL
2900.INCMEM
2910INC MEMLOC3+1
2920INC MEMLOC4+1
2930BNE NTZERO
2940INC MEMLOC2+1
2950INC MEMLOC4+2
2960.NTZERO
2970DEX
2980BNE  INCMEM
2990DEY
3000BNE  MEMLOC2
3010JSR  OSNEWL
3020RTS
3030.BYTE
3040PLA
3050STA  RETADRL1+1
3060PLA
3070STA  RETADRH1+1
3080PLA
3090TAX
3100PLA
3110JSR HEXCON
3120ASLA
3130ASLA
3140ASLA
3150ASLA
3160STA MEMLOC5+1
3170TXA
3180JSR HEXCON
3190.MEMLOC5
3200ORA #0 Dummy
3210TAX
3220.RETADRH1
3230LDA #0 Dummy
3240PHA
3250.RETADRL1
3260LDA #0 Dummy
3270PHA
3280RTS
3290.HEXCON
3300CMP #&40
3310BCC LESS
3320SBC #7
3330.LESS
3340SEC
3350SBC  #&30
3360RTS
3370.HEXASC
3380PHA
3390LSRA
```

```
3400LSRA
3410LSRA
3420LSRA
3430JSR CONVT
3440PLA
3450AND #&0F
3460JSR CONVT
3470RTS
3480.CONVT
3490SED
3500CLC
3510ADC #&90
3520ADC #&40
3530CLD
3540JSR OSWRCH
3550RTS
3560.MISTAKE
3570LDA #11
3580JSR OSWRCH
3590 LDA #ASC("?")
3600JSR OSWRCH
3610LDA #7
3620JSR OSWRCH
3630JSR OSNEWL
3640RTS
3650.INBYTE
3660LDX #(TEXT7-TEXT)
3670JSR OUTPUT
3680JSR HEXINPUT
3690CPY #3
3700BEQ OK1
3710.PULL1
3720PLA
3730DEY
3740BNE PULL1
3750JSR MISTAKE
3760JMP INBYTE
3770.OK1
3780PLA
3790JSR BYTE
3800RTS
3810.INCHAR
3820JSR OSRDCH
3830CMP #&1B
3840BNE NTESC
3850LDA #&7E
3860JSR OSBYTE
3870JMP INCHAR
3880.NTESC
3890 RTS
3900.TEXT:]
3910LGTH=0:TEXT1=FNTEXT("STACK")
3920TEXT2=FNTEXT("  SP X  Y  A  NV-BDIZC")
3930TEXT3=FNTEXT("ZERO PAGE")
3940TEXT4=FNTEXT("C CONTINUE E END M MEMORY")
3950TEXT5=FNTEXT("A ALTER R REGISTERS G GO")
3960TEXT6=FNTEXT("ADDRESS (4 DIGITS)?")
3970 TEXT7=FNTEXT("ALTER TO (2 DIGITS)?")
3980TEXT8=FNTEXT("WHICH (X,Y,A,P)?")
3990NEXTI%
4000VDU12:PRINT"**************BBC MONITOR***************"'
4010PRINT"Access from assembler with JSR S%"''"Access
    from BASIC with CALL S%"
4020*KEY10"OLD|MHIMEM=HIMEM-&330|M"
4030END
4040DEF FNTEXT(A$)
4050L=LGTH:LGTH=LGTH+LEN(A$)+1
4060$(TEXT+L)=A$
4070=TEXT+L
```

Run the program and enter the monitor with CALL S% (CALL HIMEM would do just as well if you ever want to use S%). The first heading is a display of the stack just prior to the instruction which called the monitor. Here, the stack is empty (apart from the two byte return address which was deposited when CALL entered the monitor).

Next we see the registers, with each flag of the status register visible. We have not displayed the program counter for reasons I shall explain in a moment.

The contents of &70 to &8F is the final display, followed by the menu. C is used in conjunction with the JSR S% entry, which we will consider in a moment. E returns to BASIC at the point of a CALL (but not a USR) instruction. M displays any section of memory; and A allows you to alter any section of memory. R allows you to alter X, Y, A or P. G allows you to go to any subroutine you like. Pressing any other letter will do nothing. However, you can press CTRL-N (page mode on), CTRL-O (page mode off), CTRL-B (printer on) and CTRL-C (printer off) and it will be accepted.

Let us now look at M, A, R and G in more detail. Press M, and you are asked for the starting address of the 32 bytes to be displayed. This address must be given as 4 hex digits (so that even 0 must be given as 0000). If you type in any number of digits apart from 4, an indication of error will be displayed: try it. If you attempt to input a non-digit it will not be accepted (this is true of A, R and G also): try this too. Finally, if you realise you pressed M by mistake you can return to the display (from the registers onwards) by pressing carriage return immediately. This will work with A, R and G too. Try some addresses now, and display their contents.

Pressing A will allow you to alter the contents of any memory location. Again 4 digits exactly are required for the address, and 2 for the contents. Try 0070, and alter it to 8A; then 0071 and alter to A8; and then 0072 altering it to 60. Notice how, if it is user zero page you are altering (which is the most usual), then the altered contents are shown immediately.

Pressing R will allow you to alter X, Y, A or P. Let us alter A to 50, X to E3 and P to 81. Notice that these register contents are only activated when you use C or G. Let us try G: go to 0070, which now contains TXA: TAY: RTS and see what happens. Is this what you expected? One final experiment in this phase: alter 01FE to CF, 01FF to 78 and type E. Why does this happen? Press the break key to get out of this, and notice the effect of line 4020.

Let us now examine how we can use the monitor to aid debugging. Load Listing 10.5 into the computer now (make sure you have a copy of Listing 10.6 on tape) and insert this line: 335 JSR S%. Type RUN, and search for the bytes E9 00. You can follow the program through LOOP1 by pressing C after you have inspected the display in the monitor: if you want to, you can change any locations you like. In this way, you can easily see if the logic of the problem is what you expect it to be.

On first entry, the stack is B3 8E; this holds the return address to the CALL at line 210 (which is &8EB4). Zero page holds the correct data, with MEMTOTAL and SEARCHBASE both &8000 and BYTECOUNT with 2. At this stage, MATCHBASE is immaterial, since the accumulator indicates 4C as the current item of memory. Press C again: now Y is 2, and A is IF. Press C again twice giving Y:3, A:80; and Y:4, A:4C. Now press C again and look at MATCHBASE: it is &8004, so E9 must have been located here. A is 4 since it contains the old value of Y (from line 730). Type M and then &8000 to check the operation is correct.

Now use A to alter &8E to BE and &8F to D0 and use R to set Y to 01. Press C, and see what happens. There has been a match, but print-out was too rapid to see; but X is equal to BYTECOUNT and this is the clue. Use M to verify that DDBF and DDC0 contain E9 00.

Let us monitor the printout operation now: delete 335, and put JSR S% at 1005. Search again for E9 00. We enter the monitor when MATCHBASE is &864A. Let us examine the stack: EA 17 is the return address to line 920; 86 is the high byte of the match (put on at 860); D5 17 is the return address to line 780; 4A is the value of Y at the beginning of the match; and 93 17 is the return address to line 330. We note that A contains 38, the ASCII code for 8. C again puts 36 in the accumulator (ASCII for 6). Press C twice more, and check the registers, especially A and P. Continue in this way to monitor the printout of the address &8673.

You should now have a good idea how the monitor can help you to debug programs. By putting JSR S% into sensible points you can quickly determine a fault. You will find this monitoring process much easier if you have a copy of the program being monitored (a printer is very useful in assembly code work, but copying out by hand is not very laborious except for programs of the monitor's length). Indicate by pencil on your copy where your JSR S% has been put, so you can follow through the logic. Since putting in a JSR S% upsets the address location of the program, there is little advantage of following the contents of the PC through, and so little advantage in displaying the PC. You can, of course, use more than one JSR S% at the same time, but in many cases, you will find it easier to take each small section of the program separately with just one JSR S%. However, there are times when you want to single-step through some small section of code, and this is easily done by putting a JSR S% after each line you want to monitor. Don't be tempted to single-step through large sections of code, however. It is a very slow process, and is not usually very revealing. By using your intimate knowledge of the program being debugged, you will be able to pinpoint the area where monitoring will be most efficacious.

When you write assembly programs of even moderate complexity you will often find that they don't work first time! The break key is a boon at these moments (which is why line 4020 programs it to safeguard the relevant locations). The monitor too will be very useful to you, but it should not be used as a first resort. There are some simple checks you should always make first. In order of regularity they are:

(a) Missing off the # in immediate statements e.g. LDA 10 instead of LDA #10.

(b) Missing off the & for hex e.g. STA 70 instead of STA &70.

(c) Forgetting to clear or set carry in ADC or SBC instructions.

(d) Forgetting to save a result after addition or subtraction e.g. LDA &70: CLC: ADC #1: BCC NOCARRY: LDA &71 ..... etc. Here, we have forgotten to put STA &70 after the ADC #1.

(e) Forgetting that INC and DEC do not put anything into the accumulator.

(f) Forgetting that CMP, CPX and CPY affect the carry flag.

(g) Using BMI and BPL where BCC and BCS should be used.

(h) Missing off the NEXT in a two-pass assembly (so only getting one pass).

(i) Not allocating enough room in DIM START statements.

If none of these are the cause, then is the time to use the monitor. Pay particular attention to the indexes in arrays and in indirect indexed addressing: monitor code containing these first. Be patient, systematic and sensible and debugging will not take you long: approach it randomly, hoping for good luck, and it could take you ages.

Let us finish now by looking at the details of the program itself. Notice in particular the use of the function at 4040 to 4070: its purpose is to allow us to use multi-outputs (here there are eight) without having to calculate their lengths. We could change the contents of any of lines 3910 to 3980 and no other changes will be required to the program.

The program is written so as to be as self-contained as possible. In particular, apart from the stack and registers, no memory locations outside the space of the program itself are used. Hence, the monitor cannot corrupt any memory locations, a very important consideration. Only two system routines are needed (OSWRCH and OSRDCH), and since these are provided on all computers, the program can, with very little modification, be used as a monitor on any 6502 computer.

The program details are:

70-120 Put registers on the stack in the order PAYX.

130-140 Clear decimal and interrupt flags (if set they will be re-set when P is retrieved from the stack).

150-160 Clear screen.

170-180 Put heading "STACK".

190-230 Save adjusted stack pointer on the stack (adjusted by 6 to skip over the four pushes in 70-120 plus the two byte return address).

240-250 Point to last item on the stack before entry to MONITOR, and put in X register. (Remember that SP points to next free location - hence the need for INX.)

260 Jump to 370 if stack empty.

280-350 Otherwise print out contents of stack in hex. Leave three spaces between each item.

370-390 Output a new line and print out register headings.

400-520 Print out contents of SP, X, Y and A under headings.

530-660 Output 8 bits of P in the order most significant to least significant. Prior to line 600, the accumulator always contains ASCII zero. If C = 1, this is replaced in 610 by ASCII one.

670-690 Output a new line and print "ZERO PAGE".

700-760 Pass parameters to subroutine MEMOUT (at 2680). These will cause 32 bytes to be output from location &0070 onwards.

780-810 Print out menu.

830 Get a character from the keyboard (at 3820).

840-940 Allow through page and printer control codes.

950-990 If E pressed, set stack pointer to point to return to BASIC (i.e. first two bytes on the stack) and return.

1010-1100 If C pressed, restore registers in order X, Y, A, P and return to calling program. (1030 throws away the adjusted stack pointer put on stack at 190-230).

1120-1170 Branch to relevant section if M, G or R.

1180-1190 If not A, return to 830.

1200-1270 Coding if A pressed.

1200 Go to 'get address' routine at 1870 (used by M or A).

1210 Put low byte of address (stored in X after return from MORA) in low byte position of line 1260.

1220-30 Put high byte of address (in X from BYTE at line 3040) in high byte position of 1260.

1240 Get a byte from the keyboard (at 3660).

1260-70 Put keyboard byte (stored in X) in address defined by 1210-1230, and return to zero page display etc.

1290-1360 Coding if M pressed.

1290-1310 Go to 'get address' routine at 1870; store low byte in 2820 and in 2740.

1320-1340 Store high byte in 2820 and in 2710.

1350-1360 Display 32 bytes from the address specified in 1290-1340, and return to the menu.

1380-1510 Coding if G pressed

1380-90 Go to 'get address' routine at 1870; store low byte in 1500.

1400-1410 Store high byte in 1500.

1420-80 Restore registers (as 1010-1100).

1500-1510 Jump to subroutine specified in 1380-1410 and on return go to the start of the program.

1530-1840 Coding if R pressed.

1530-40 Print out "WHICH (X,Y,A,P)?"

1560-80 Get a character from keyboard (at 3820). If carriage return, jump to display via 1850.

1590 Put stack pointer plus 2 into X (plus 2 to skip over adjusted stack pointer).

1620-1720 Registers in stack are in the order X, Y, A, P. Increment X to point to relevant point on the stack or else return to 1560.

1740 Output the letter X, Y, A or P typed in.

1750-1770 Save X register on stack (used in INBYTE), and go to get a byte from the keyboard.

1780-1820 Achieves the result X → A, top of stack → X.

1830-1840 Put new value of register (now in A) in relevant position in the stack (pointed to by X set in 1620-1720) and return to display.

1860-3890 Subroutines.

1860 The 'get address' routine used by M or A.

1870-1900 Save return address in lines 2120 and 2150. This allows stack to be used to pass parameters on return.

1920-1930 Print "ADDRESS" (4 DIGITS)?"

1940 Go to routine which puts digit plus CR onto stack (at 2190).

1950-60 If five characters, input is OK.

1970-2000 If just CR, pull off stack and return to display.

2020-2060 If there aren't five characters, pull them all off the stack, signal the mistake (at 3570) and ask for address again (at 1920).

2080-2170 If five characters, throw CR away from the stack, output a new line, translate low byte of address from ASCII to hex (at 3040), put return address on the stack (stored from 1880-1900) and return.

2180 Routine to input a series of hex digits in ASCII from the keyboard.

2190-2220 As 1870-1900, but store in 2530 and 2560.

2240 Set digit count to zero.

2260-2300 Get a character from the keyboard. If delete or CR, skip over check for a hex digit.

2310-2340 If less than &30 or more than &46, cannot be a digit.

2350-2360 If less than &3A will be 0 to 9.

2370-80 If more than &40 will be A - F.

2400 If character is suitable (i.e. it passes tests in 2260-2390) print it out (giving a new line if CR).

2410-2460 If delete, throw away last member input to the stack (2440 checks that input not empty). Decrement the character count (in Y) and return to 2260 for next character.

2480-2580 If not delete, increment character count. If CR, return from subroutine after restoring return address. Otherwise return to 2260 for another digit.

2600-2670 Routine to output string whose starting address is indexed by X. The CR is output also.

2680 Routine to display 32 bytes. Starting address of bytes is passed to line 2820 prior to entry. The low byte is also passed to 2740 and the high byte to 2710 prior to entry.

2710-2790 Display address of current 8 bytes followed by a colon and a space.

2800-2890 Display current 8 bytes, terminated by a new line.

2910-1980 Increment low bytes in lines 2740 and 2820 by 8, incrementing by one high byte of 2820 and 2710 if necessary.

2990-3020 Repeat for four sets of 8 bytes, output a new line and return.

3030 Routine to convert a byte in ASCII to a hex byte.

3040-3070 As lines 1870-1900 except store in 3230 and 3260.

3080-3100 Put top of stack in X (this is the top hex digit of the byte), and next item of stack in A (the bottom hex digit of the byte).

3110-3160 Convert bottom ASCII digit to hex digit (at 3300), shift bottom nybble to top nybble and store temporarily in 3200.

3170-3210 Convert top ASCII digit to hex digit, combine with bottom digit already at 3200 and transfer to X.

3220-3280 Return.

3290 Routine to convert single ASCII digit to hex.

3300-3320 If more than &40 (i.e. A - F) subtract 7 e.g. A (= &41) → &3A (= &30 + &0A).

3340-60 Subtract &30 to give hex digit in accumulator and return.

3370-3550 Displays byte on the screen (see 850-1030 of Listing 10.5).

3560 Give visual and aural indication of an error.

3570-3630 Move cursor up, display a question mark, output a short bleep and a new line, and return.

3650 Routine to accept a single byte from the keyboard and convert it to hex.

3660-3670 Print "ALTER TO(2 DIGITS)"

3680 Get a series of hex digits in ASCII from the keyboard (at 2190).

3690-3760 If not 3 characters, pull characters off the stack, indicate error (at 3570) and go back to 3660.

3780-3800 Otherwise, remove CR from stack, convert byte to hex (at 3040) and return.

3810 Accept a character from the keyboard.

3820-3890 Get a character. If ESCAPE, acknowledge with &7E through OSBYTE and ignore. Otherwise return.

## Exercise 10

(So that you can prove to yourself how much you have learnt, no solutions will be provided for this exercise. That your programs work will be proof enough!)

1. Modify RETRIEVE so that instead of putting FF immediately, it first checks if the next line would have had a valid line number (i.e. greater than the last one). If so, it then searches the next 250 bytes for 0D. If it comes across any of the bytes 00 to 1F it terminates and puts FF as before. But if it finds 0D, it modifies the line length byte, accepts the line, and continues to examine the next one.

2. Modify INTSORT and STRINGSORT so that a flag is set to zero at the start of each scan through the data, and set to one if any swop is made during that scan. Thus, if at the end of any scan the flag is still zero, the sort can be terminated. Do you still need to check that NUMBER equals zero with this modification? Under what conditions is the sort speeded up significantly by this change?

3. Modify REMSPACE so that:

(i) it deals with multi-line assembler statements e.g.

```
LOOP LDA #3: LOOP1 LDX #0
```

(ii) it removes assembler comments e.g.

```
LDA HIGH / Load high byte: LDX LOW
```

(Remember that unlike REM, / does not cause all later text to be ignored by the assembler, and that in some cases / can be replaced by a space.)

(iii) it deals with line number references to REM statements.

4. Incorporate the byte-searching routine of MEMORYHUNT into the MONITOR, with suitable commands added to the menu.

# Answers to Exercises

## Exercise 1

2. (i) The contents need to change.

   (ii) The address must be permanently available, even when power is off.

3. There is a limit to the number of pins economically available in a package. Once 40 pins became established, it became very difficult to produce other size packages. Another problem is that, until recently, a 16-bit bus would be too slow - however, 16-bit processors are now a reality.

4. Data can go *into* memory and *out of* memory. A memory location is chosen, however, by sending an address *towards* it. No *address* information needs to come *from* it.

5. Because each instruction needs to be translated every time it is executed in an interpreted language. With compilation the translation occurs just once, prior to execution.

## Exercise 2

1. (a) Immediate

| &0D00 | &0D01 |
|---|---|
| A9 | 0E |

(b) Absolute

| &0D00 | &0D01 | &0D02 |
|---|---|---|
| AD | 40 | 7F |

(c) Zero page

| &0D00 | &0D01 |
|---|---|
| A5 | 20 |

(d) Absolute

| &0D00 | &0D01 | &0D02 |
|---|---|---|
| 8D | 72 | 7A |

(e) Zero page

| &0D00 | &0D01 |
|---|---|
| 85 | 00 |

(f) Immediate

| &0D00 | &0D01 |
|---|---|
| A9 | 12 |

(g) Zero page

| &0D00 | &0D01 |
|---|---|
| 85 | 02 |

(h) Zero page

| &0D00 | &0D01 |
|---|---|
| A5 | 0E |

(i) Absolute

| &0D00 | &0D01 | &0D02 |
|---|---|---|
| 8D | 00 | 10 |

2. The contents of NUM1 is already in the accumulator.

```
P%   = &0D00
NUM1 = &70
NUM2 = &71
LDA    #17
STA    NUM1
STA    NUM2
```

```
&0D00    &0D01    &0D02    &0D03    &0D04    &0D05
  A9       11       85       70       85       71
```

3.

```
P%   = &0D00
NUM1 = &70
NUM2 = &71
NUM3 = &73
LDA    NUM3
STA    NUM1
LDA    NUM2
STA    NUM3
LDA    NUM1
STA    NUM2
```

## Exercise 3.1

```
 10NUM1=&70:NUM2=&71:NUM3=&72:SUML=&73:SUMH=&74
 20DIM P% 50
 30[OPT3
 40.START
 50LDA NUM1
 60CLC
 70ADC NUM2
 80STA SUML
 90LDA #0
100STA SUMH
110ADC SUMH
120STA SUMH
130LDA SUML
140CLC
150ADC NUM3
160STA SUML
170LDA #0
180ADC SUMH
190STA SUMH
200RTS:]
210REPEAT
```

```
 220INPUT"First number to be added",?NUM1
 230INPUT"Second number to be added",?NUM2
 240INPUT"Third number to be added",?NUM3
 250CALLSTART
 260PRINT?NUM1+?NUM2+?NUM3,256*?SUMH+?SUML
 270UNTIL FALSE
```

Exercise 3.2

1.

```
 10NUM1L=&70:NUM1H=&71:NUM2L=&72:NUM2H=&73:SUM0=&74:
   SUM1=&75:SUM2=&76
 20DIM P% 50
 30[OPT3
 40.START
 50LDA NUM1L
 60CLC
 70ADC NUM2L
 80STA SUM0
 90LDA NUM1H
100ADC NUM2H
110STA SUM1
120LDA 30
130STA SUM2
140ADC SUM2
150STA SUM2
160RTS:]
170REPEAT
180INPUT"First number to be added",!NUM1L
190INPUT"Second number to be added",!NUM2L
200CALLSTART
210PRINT256*?NUM1H+?NUM1L+256*?NUM2H+?NUM2L,65536*?SUM2+256
   *?SUM1+?SUM0
220UNTIL FALSE
```

2.

```
 10DIM NUM1(3),NUM2(3),RESULT(3)
 20FOR I%=0 TO 3:NUM1(I%)=&70+I%
 30NUM2(I%)=&74+I%:RESULT(I%)=&78+I%:NEXTI%
 40DIM P% 50
 50[OPT3
 60.START
 70LDA NUM1(0)
 80CLC
 90ADC NUM2(0)
100STA RESULT(0)
110LDA NUM1(1)
120ADC NUM2(1)
130STA RESULT(1)
140LDA NUM1(2)
150ADC NUM2(2)
160STA RESULT(2)
170LDA NUM1(3)
180ADC NUM2(3)
190STA RESULT(3)
200RTS:]
210REPEAT
220INPUT"First number to be added",!NUM1(0)
230INPUT"Second number to be added",!NUM2(0)
240CALLSTART
250PRINT!NUM1(0)+!NUM2(0),!RESULT(0)
260UNTIL FALSE
```

Symbolic representation is:

```
(NUM1(3); NUM1(2); NUM1(1); NUM1(0)) + (NUM2(3); NUM2(2);
 NUM2(1); NUM2(0)) → RESULT(3); RESULT(2); RESULT(1); RESULT(0)
```

Exercise 3.3

1.

```
LDA NUM1L
SEC
SBC NUM2L
STA DIFFL
LDA NUM1H
SBC NUM2H
STA DIFFH
```

A mathematical demonstration of why this works may be helpful to you. In general, for *any* numbers A and B, $A - B = A + B_c - B_c - B$, where $B_c$ is the two's complement of B, $= (A + B_c) - (B_c + B) = A + B_c$, if we ignore the 'carry'.

Now, if A and B are double bytes, so that $A = (AH;AL)$ and $B = (BH;BL)$ then $B_c = (\overline{BH};BL_c)$, where $\overline{BH}$ is the one's complement of BH. Hence, $A - B = (AH + \overline{BH} + C; AL + BL_c)$, where C is the carry. Now, $AL + BL_c$ is achieved by the first three lines of the program. If this is positive C will be one, if negative C will be zero. Either way, the second part of the program performs $AH + \overline{BH} + C$ as required.

2.

```
10DIM NUM1(3),NUM2(3),RESULT(3)
20FOR I%=0 TO 3:NUM1(I%)=&70+I%
30NUM2(I%)=&74+I%:RESULT(I%)=&78+I%:NEXTI%
40DIM P% 50
50[OPT3
```

```
 60.START
 70LDA NUM1(0)
 80SEC
 90SBC NUM2(0)
100STA RESULT(0)
110LDA NUM1(1)
120SBC NUM2(1)
130STA RESULT(1)
140LDA NUM1(2)
150SBC NUM2(2)
160STA RESULT(2)
170LDA NUM1(3)
180SBC NUM2(3)
190STA RESULT(3)
200RTS:]
210REPEAT
220INPUT"First number",!NUM1(0)
230INPUT"Number to be subtracted",!NUM2(0)
240CALLSTART
250PRINT!NUM1(0)-!NUM2(0),!RESULT(0)
260UNTIL FALSE
```

Exercise 3.4

1. (a) &18 - &EE = &18 + &12 = &2A    (= 42)

   (b) &AA - &23 = &AA + &CC = $\boxed{1}$&87  (= -121)

   Overflow can only occur if the signs of the numbers subtracted are different.

2. $-2^{31}$ to $2^{31} - 1$

3. No adjustments necessary.

   IF SUM(3) > 127 THEN RESULT = (((SUM(3) - 255) * 256 + (SUM(2) - 255)) * 256 + (SUM(1) - 255)) * 256 + SUM(0) - 256

   ELSE RESULT = ((SUM(3) * 256 + SUM(2)) * 256 + SUM(1)) * 256 + SUM(0)

Exercise 3.5

(a) ORA #&88
    AND #&EE

(b) EOR #&80
    ORA #&40
    AND #&E0

Exercise 4.1

1.

```
                LDA NUM1
                CLC
                ADC NUM2
                BEQ ZERO
                BPL POSITIVE
       ZERO          .
                     .
                     .
                     .
                     .
       POSITIVE      .
                     .
                     .
                     .
```

2.

```
            LDA NUM1
            SEC
            SBC NUM2
            BEQ ZERO
            LDA NUM2
            STA NUM3
            LDA NUM1
            STA NUM2
            LDA NUM3
            STA NUM1
       ZERO     .
                .
                .
                .
                .
```

3.

```
                LDA NUM1
                CLC
                ADC NUM2
                BCC NOCARRY
                LDA #0
                STA SUM
                BEQ OVER    (always branches)
       NOCARRY  STA SUM
       OVER        .
                   .
                   .
```

Congratulate yourself if you got this. Congratulate yourself even more if you got the following more economical version:

```
          LDA NUM1
          CLC
          ADC NUM2
          STA SUM
          BCC NOCARRY
          LDA #0
          STA SUM
   NOCARRY    .
              .
              .
              .
              .
```

Exercise 4.2

1.

```
          LDA NUM
          CMP #15
          BEQ LESSEQ
          BCC LESSEQ
          LDA #0
          STA NUM
   LESSEQ     .
              .
              .
              .
              .
```

2.

```
          LDA NUM
          BPL POSITIVE
          CMP #&F6
          BEQ EQUAL
          BCS POSITIVE   (or BPL POSITIVE)
 EQUAL    STA INDIC
          JMP OVER
 POSITIVE LDA #0
          STA NUM
          LDA #1
          STA INDIC
```

```
OVER            .
                .
                .
                .
                .
```

```
3.              LDA NUM1
                CMP NUM2
                BNE NTEQUAL1
                LDA NUM3
                STA NUM2
                JMP OVER
      NTEQUAL1  LDA NUM4
                CMP #16
                BEQ NTEQUAL2
                BCC NTEQUAL2
                STA NUM2
                JMP OVER
      NTEQUAL2  LDA #0
                STA NUM2
                STA NUM4
      OVER          .
                    .
                    .
                    .
                    .
```

Exercise 4.3

```
1.              LDA NUM1L
                CMP NUM2L
                LDA NUM1H
                SBC NUM2H
                BCC LESS
                    .
                    .
                    .
                    .
      LESS          .
```

```
2.              LDA NUM1L
                CMP NUM2L
```

```
             BNE NTEQUAL
             CLC
    NTEQUAL  LDA NUM1H
             SBC NUM2H
             BCC LESSEQ
                 .
                 .
                 .
                 .
    LESSEQ       .
```

```
3.           LDA NUM1(0)
             CMP NUM2(0)
             LDA NUM1(1)
             SBC NUM2(1)
             LDA NUM1(2)
             SBC NUM2(2)
             LDA NUM1(3)
             SBC NUM2(3)
             BCS GTEQUAL
                 .
                 .
                 .
                 .
    GTEQUAL      .
```

```
4.           LDA NUM(0)
             ORA NUM(1)
             ORA NUM(2)
             ORA NUM(3)
             BEQ ZERO
                 .
                 .
                 .
                 .
    ZERO         .
```

The accumulator will at the end contain ones in the positions where ones occur in any of the four bytes tested. Hence, only if all four bytes are zero will the accumulator be zero.

Exercise 4.4

```
(a)          LDA NUM1L
             SEC
             SBC NUM2
             STA NUM1L
```

```
                LDA NUM1H
                SBC #0
                STA NUM1H

(b)             LDA NUM1L
                SEC
                SBC NUM2          Three bytes are saved.
                STA NUM1L
                BCS NOBORROW
                DEC NUM1H
      NOBORROW    .
                  .
                  .
                  .
                  .
```

Exercise 4.5

```
1.              LDA NUM1
                SEC
                SBC NUM2
                BCC LESSEQ
                CMP NUM3
                BEQ LESSEQ
                BCS GREATER
      LESSEQ      .
                  .
                  .
                  .
                  .
      GREATER     .

2.              LDA #0
                STA DIFF2
                LDA NUM1L
                SEC
                SBC NUM2L
                STA DIFF0
                LDA NUM1H
                SBC NUM2H
                STA DIFF1
```

```
         BCS NTNEG
         DEC DIFF2
NTNEG        :
```

3.

```
         LDA #0
         STA SUMH
         LDA NUM1
         CLC
         ADC NUM2
         STA SUML
         BVC NOOVFLOW
         EOR #&80
NOOVFLOW BPL NTNEG
         DEC SUMH
NTNEG        .
             .
             .
             .
             .
```

4.

```
         LDA #0
         STA DIFF2
         LDA NUM1L
         SEC
         SBC NUM2L
         STA DIFF0
         LDA NUM1H
         SBC NUM2H
         STA DIFF1
         BVC NOOVFLOW
         EOR #&80
NOOVFLOW BPL NTNEG
         DEC DIFF2
NTNEG        :
```

5.

```
         LDA NUM1L
         SEC
         SBC NUM2L
         STA NUM1L
```

```
                 LDA NUM1M
                 SBC NUM2H
                 STA NUM1M
                 BCS NOBORROW
                 DEC NUM1H
                 BPL OVERFLOW
        NOBORROW      .
                      .
                      .
                      .
                      .
        OVERFLOW      .
```

Overflow occurs here if NUM1H goes from &80 to &79, and the N flag will register this. The V flag is not affected by DEC or by INC, since the N flag tells us exactly what we want to know. The OVERFLOW routine here would probably be designed to return some error message.

```
6.               LDA #0
                 STA SUMH
                 LDA NUM1L
                 CLC
                 ADC NUM2L
                 STA SUML
                 LDA NUM1H
                 ADC NUM2H
                 STA SUMM
                 BVC NOOVFLOW
                 EOR #&80
        NOOVFLOW BPL NTNEG
                 DEC SUMH
        NTNEG         .
                      .
                      .
                      .
                      .
```

```
7.               LDA NUM1
                 SEC
                 SBC NUM2
```

```
          BVC NOOVFLOW
          EOR #&80
NOOVFLOW BPL GTEQUAL
               .
               .
               .
               .
GTEQUAL        .
               .
               .
```

```
8.        LDA NUM1
          SEC
          SBC NUM2
          BVC NOOVFLOW
          BPL LESSEQ  (Since overflow occurred, this branches
                       if the result is negative i.e.
                       NUM1 - NUM2 < 0
          BMI OVER    - and this branches if result positive)
NOOVFLOW BMI LESSEQ
OVER      CMP NUM3    Usual comparison between unsigned
                      numbers.
          BEQ LESSEQ
          BCS GREATER
LESSEQ         .
               .
               .
               .
               .
GREATER        .
```

```
9.        LDA NUM1
          SEC
          SBC NUM2
          BVC NOOVFLOW1
          BMI MORE    Positive overflow must exceed NUM3.
          BPL LESSEQ  Negative overflow must be less than
NOOVFLOW1 SEC         NUM3.
          SBC NUM3
          BEQ LESSEQ
          BVC NOOVFLOW2
          EOR #&80
```

```
NOOVFLOW2 BPL MORE
LESSEQ        .
              .
              .
              .
              .
MORE          .
              .
              .
              .
              .
```

```
10.            LDA NUM1L
               CMP NUM2L
               LDA NUM1H
               SBC NUM2H
               BVC NOOVFLOW
               EOR #&80
      NOOVFLOW BPL GTEQUAL
                   .
                   .
                   .
                   .
      GTEQUAL      .
```

Exercise 5.1

1. Yes they will. For example, FOR X = -2 TO 50 in (e) will result in just one cycle of the loop since it will be understood as FOR X = 254 TO 50.

   We solve such problems by using BPL and BMI instead of BCS and BCC after comparisons (so that, if NUM is signed, it is the second program in (d) which is correct). It is not quite this simple in (f), however, since we can get overflow:
   FOR X = 126 TO 10 STEP 20 is an example; FOR X = 50 TO 100 STEP 40 is another.

   Hence we need a test for overflow also, and the solution to question 3 deals with this.

```
2. (a)        LDX NUM2                 (b)        LDX NUM2
     LOOP         .                         LOOP      .
                  .                                   .
                  .                                   .
                  .                                   .
                  .                                   .
              DEX                                 TXA
```

```
        CPX #&FF
        BEQ OUT
        CPX NUM1
        BCS LOOP
OUT         .
            .
            .
            .
            .
```

```
        SEC
        SBC NUM3
        BCC OUT
        TAX
        CMP NUM1
        BCS LOOP
OUT         .
            .
            .
            .
            .
```

```
3.          LDX NUM1
      LOOP    .
              .
              .
              .
              .
            TXA
            CLC
            ADC NUM3
            TAX
            BVS OUT     If overflow, loop must be finished.
            CMP NUM2
            BMI LOOP
            BEQ LOOP
      OUT     .
              .
              .
              .
              .
```

It is quicker still to work 'backwards' (although - (NUM3) may turn out to be positive):

```
            LDX NUM2
      LOOP    .
              .
              .
              .
              .
            TAX
            SEC
            SBC NUM3
```

```
          TAX
          BVS OUT
          CMP NUM1
          BPL LOOP
     OUT     .
             .
             .
             .
             .
```

## Exercise 5.2

```
1. After LDX #0 put LDA NUMH
                    BEQ LOLOOP
```

2. (i) In this case, all that is required is that we create a loop with (NUMH; NUML) + 1 cycles. The most efficient way to do this is:

```
          LDX NUML
   LOOP1     .
             .
             .
             .
             .
          DEX
          CPX #&FF
          BNE LOOP1
          LDY NUMH
          BEQ OUT
          INX
   LOOP2     .
             .
             .
             .
             .
          DEX
          BNE LOOP2
          DEY
          BNE LOOP2
   OUT       .
             .
             .
             .
             .
```

(ii) N cannot be computed in any simple way from (i), since N does not decrease in strict descending order (in LOOP2 &00 precedes &FF). Also the high byte of N is given by Y - 1 in LOOP2.

We require a loop in strict order where N is (Y;X) in LOOP2 and (NUMH;X) in LOOP1:

```
        LDX NUML
LOOP1    .
         .
         .
         .
         .
        DEX
        CPX #&FF
        BNE LOOP1
        LDY NUMH
        BEQ OUT
        DEY
LOOP2    .
         .
         .
         .
         .
        DEX
        CPX #&FF
        BNE LOOP2
        DEY
        CPY #&FF
        BNE LOOP2
OUT      .
         .
         .
         .
         .
```

Only (i) is an improvement over the forward loop.

Exercise 5.3

```
1. (a)     TXA                 (b)          STX MEMLOC+1
           SEC                              SEC
           SBC M                     MEMLOC SBC #0
           STA M
```

```
(c)           TXA                   (d)           STX MEMLOC+1
              SEC                                 SEC
              SBC M                        MEMLOC SBC #0
              TAX                                 TAX

(e)           TXA                   (f)           TXA
              STY MEMLOC+1                        STY MEMLOC+1
              CLC                                 SEC
       MEMLOC ADC #0                       MEMLOC SBC #0

(g)           STA MEMLOC+1          (h)           STY MEMLOC+1
       MEMLOC CPX #0                       MEMLOC CPX #0

(i)           STX MEMLOC+1
              TAX
       MEMLOC LDA #0
```

```
2.            STA MEMLOC+1
              TXA
              CLC
              ADC M
              STA M
       MEMLOC LDA #0
```

```
3.            STA MEMLOC+1                        STX MEMLOC1 + 1
              TXA                                 STA MEMLOC2 + 1
              SEC                                 SEC
              SBC M                       MEMLOC1 SBC #0
              TAX                                 TAX
       MEMLOC LDA #0                      MEMLOC2 LDA #0
```

```
4.            STY MEMLOC1 + 1
              STA MEMLOC2 + 1
              TXA
              CLC
      MEMLOC1 ADC #0
              TAX
      MEMLOC2 LDA #0
```

*NB*. The use of the stack makes Qus.2, 3 and 4 easier to solve. We will return to this in Chapter 9.

## Exercise 5.4

1.

```
 10TERML=&70:TERMH=&71:NUM=&72:SUM1=&73:SUM2=&74:
   SUM3=&75:?&76=0
 20DIM START 100
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60TAX
 70STA SUM2
 80STA SUM3
 90STA TERMH
100LDA #1
110STA SUM1
120STA TERML
130.LOOP
140INX
150CPX NUM
160BEQ FINISH
170STX MEMLOC+1
180CLC
190LDA TERML
200.MEMLOC
210ADC #0 Dummy operand
220STA TERML
230BCC NOCARRY
240CLC
250INC TERMH
260.NOCARRY
270LDA TERML
280ADC SUM1
290STA SUM1
300LDA TERMH
310ADC SUM2
320STA SUM2
330BCC LOOP
340INC SUM3
350BCS LOOP
360.FINISH
370RTS:]NEXTI%
380CLS:REPEAT
390INPUT"How many terms",?NUM
400CALLSTART
410PRINT!SUM1
420UNTIL FALSE
```

Lines 140-160 stop the loop being entered if (NUM) = 1.
Lines 170-250 compute the (X + 1)th term (Xth term + X).
Lines 270-350 add the (X + 1)th term to the sum for X terms to
get the sum for (X + 1) terms.

2.

```
 10TERML=&70:TERMH=&71:SUM1=&72:SUM2=&73:SUM3=&74
   :STOTAL1=&75:STOTAL2=&76:STOTAL3=&77
 20DIM START 100
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60TAX
 70STA SUM2
 80STA SUM3
 90STA TERMH
100LDA #1
110STA SUM1
120STA TERML
130.LOOP
140INX
150LDA STOTAL1
160CMP SUM1
170LDA STOTAL2
180SBC SUM2
190LDA STOTAL3
200SBC SUM3
210BCC FINISH
220STX MEMLOC+1
230CLC
240LDA TERML
250.MEMLOC
260ADC #0 Dummy operand
270STA TERML
280BCC NOCARRY
290CLC
300INC TERMH
310.NOCARRY
320LDA TERML
330ADC SUM1
340STA SUM1
350LDA TERMH
360ADC SUM2
370STA SUM2
380BCC LOOP
390INC SUM3
400BCS LOOP
410.FINISH
420RTS:]NEXTI%
430CLS:REPEAT
440INPUT"Maximum total",!STOTAL1
450PRINT((USRSTART AND &0FFFFFFF)MOD &10000) DIV &100
460UNTIL FALSE
```

As Q.1 except lines 150 and 160 are replaced by 150-210.

3.

```
 10TERML=&70:TERMH=&71:SUM1=&72:SUM2=&73:SUM3=&74:STOTAL1=&75:
   STOTAL2=&76:STOTAL3=&77
 20DIM START 100
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA #0
 60TAX
 70STA SUM2
 80STA SUM3
 90STA TERMH
100LDA #1
110STA SUM1
120STA TERML
130.LOOP
140INX
150SEC
160LDA STOTAL1
170SBC SUM1
180STA MEMLOC1+1
190LDA STOTAL2
200SBC SUM2
210STA MEMLOC2+1
220LDA STOTAL3
230SBC SUM3
240BCC FINISH1
250.MEMLOC1
260ORA #0 Dummy operand
270.MEMLOC2
280ORA #0 Dummy operand
290BEQ FINISH2
300SBC SUM2
310LDA STOTAL3
320STX MEMLOC+1
330CLC
340LDA TERML
350.MEMLOC
360ADC #0 Dummy operand
370STA TERML
380BCC NOCARRY
390CLC
400INC TERMH
410.NOCARRY
420LDA TERML
430ADC SUM1
440STA SUM1
450LDA TERMH
460ADC SUM2
470STA SUM2
480BCC LOOP
490INC SUM3
500BCS LOOP
510.FINISH1
520DEX
530.FINISH2
540RTS:]NEXTI%
550CLS:REPEAT
560INPUT"Maximum total",!STOTAL1
570PRINT((USRSTART AND &0FFFFFFF)MOD &10000) DIV &100
580UNTIL FALSE
```

As Q.1 except lines 150 and 160 are replaced by 150-310 and 360 by 510-530.

## Exercise 6.1

When an early part of the new locations overlaps a later part of the old locations.

## Exercise 6.2

1.

```
 10INPUT"HOW MANY BYTES",NUMBER
 20DIM ARRAY NUMBER-1:DIM START 50
 30FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
 40FLAG=&70:TEMP=&71
 50FORI%=0 TO 2 STEP 2:P%=START
 60[OPTI%
 70.BEGIN
 80LDX #NUMBER-1
 90LDA #0
100STA FLAG
110.LOOP
120LDA ARRAY,X
130CMP ARRAY-1,X
140BCS OVER
150STA TEMP
160LDA ARRAY-1,X
170STA ARRAY,X
180LDA TEMP
190STA ARRAY-1,X
200LDA #1
210STA FLAG
220.OVER
230DEX
240BNE LOOP
250LDA FLAG
260BNE BEGIN
270RTS:]NEXTI%
280CALL START
290 FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

(a) By reversing the comparison we gain space and time since only one branch instruction is necessary.

(b) Usually, we put values into storage locations so that we do not need to reassemble the program every time we change the values concerned. In this case, however, we will want to change the base address of the array (i.e. ARRAY) and this will require reassembly. We will see a way round this in the next chapter.

2.

```
 10INPUT"HOW MANY BYTES",NUMBER
 20DIM ARRAY NUMBER-1:DIM START 50
 30FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
 40TEMP=&70
 50FORI%=0 TO 2 STEP 2:P%=START
 60[OPTI%
 70LDY #0
 80.BEGIN
 90STY MEMLOC+1
100LDX #NUMBER-1
110.LOOP
120LDA ARRAY,X
130CMP ARRAY-1,X
140BCS OVER
150STA TEMP
160LDA ARRAY-1,X
170STA ARRAY,X
180LDA TEMP
190STA ARRAY-1,X
200.OVER
210DEX
220.MEMLOC
230CPX #0 (Dummy operand)
240BNE LOOP
250INY
260CPY #NUMBER-1
270BNE BEGIN
280RTS:]NEXTI%
290CALL START
300 FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

Notice that this is less efficient since we require an extra CPY.

3.

```
10INPUT"HOW MANY BYTES",NUMBER
20DIM ARRAY NUMBER-1:DIM START 50
30FORI%=0 TO NUMBER-2 STEP4:!(ARRAY+I%)=RND:NEXTI%
40TEMP=&70
50FORI%=0 TO 2 STEP 2:P%=START
60[OPTI%
70LDY #1
80.BEGIN
90STY MEMLOC+1
100.MEMLOC
110LDX #0 (Dummy operand)
120.LOOP
130LDA ARRAY,X
140CMP ARRAY-1,X
150BCS OVER
160STA TEMP
170LDA ARRAY-1,X
180STA ARRAY,X
190LDA TEMP
200STA ARRAY-1,X
210.OVER
220DEX
230BNE LOOP
240INY
250CPY #NUMBER
260BNE BEGIN
270RTS:]NEXTI%
280CALL START
290 FOR I%=0 TO NUMBER-1: PRINT?(ARRAY+I%),:NEXT
```

Exercise 6.3

1. The flowchart is in Figure 6.7.

```
 10DIM START 50:VDU14
 20 DATUM=&70:INDIC=&71:DIM BASMEM 255
 30FOR I%=0 TO 252 STEP4:!(BASMEM+I%)=RND:NEXTI%
 40DIM TEST 256
 50X%=0:Y%=0:?INDIC=&80
 60FOR I%=0 TO 2 STEP2:P%=START
 70[OPTI%
 80STX MEMLOC+1
 90.MEMLOC
100CPY #0 (Dummy operand)
110BNE OK
120LDA INDIC
130BMI OK
140SEC
150RTS
160.OK
170LDA #0
180STA INDIC
190LDA BASMEM,Y
200INY
210CLC
220 RTS:]NEXTI%
230FOR I%=0 TO 256
240 !&404=USR(START):X%=?&405:Y%=?&406:?(TEST+I%)=?&404
250IF (?&407 AND 1) =1 THEN PRINT"ERROR AT "STR$(I%+1)
    "TH WITHDRAWAL":GOTO270
260 NEXTI%
270PRINT'"    WITHDRAWAL  QUEUE"'
```

```
280FOR I%=0 TO 255
290PRINT ?(BASMEM+I%),?(TEST+I%):NEXTI%
300VDU15
```

2.

```
 10DIM START1  50:DIM START2 50:VDU12
 20 DATUM=&70:INDIC=&71:DIM BASMEM 255
 30X%=128:Y%=0:?INDIC=&80
 40FORI%=0 TO 124 STEP4: !(BASMEM+I%)=RND:NEXTI%
 50FOR I%=0 TO 2 STEP2:P%=START1
 60[OPTI%
 70STX MEMLOC+1
 80.MEMLOC
 90CPY #0 (Dummy operand)
100BNE OK
110LDA INDIC
120BPL OK
130SEC
140RTS
150.OK
160LDA #&80
170STA INDIC
180LDA DATUM
190STA BASMEM,X
200INX
210CLC
220 RTS:]NEXTI%
230FOR I%=0 TO 2 STEP2:P%=START2
240[OPTI%
250STX MEMLOC+1
260.MEMLOC
270CPY #0 (Dummy operand)
280BNE OK
290LDA INDIC
300BMI OK
310SEC
320RTS
330.OK
340LDA #0
350STA INDIC
360LDA BASMEM,Y
370INY
380CLC
390 RTS:]NEXTI%
400REPEAT
410INPUT'"DATA",?DATUM
420IF ?DATUM<128 THEN !&404=USR(START1) ELSE !&404=USR(START2)
430X%=?&405:Y%=?&406
440IF (?&407 AND 1) =1 THEN PRINT"ERROR"
450IF X%>Y% THEN LGTH=X%-Y% ELSE IF X%<Y% THEN LGTH=
   256-Y%+X% ELSE IF ?INDIC = 0 THEN LGTH=0 ELSE LGTH=256
460PRINT"HEAD  "?(BASMEM+Y%)
```

```
470PRINT"TAIL   "?(BASMEM+X%-1)
480PRINT"LENGTH"LGTH
490UNTIL FALSE
```

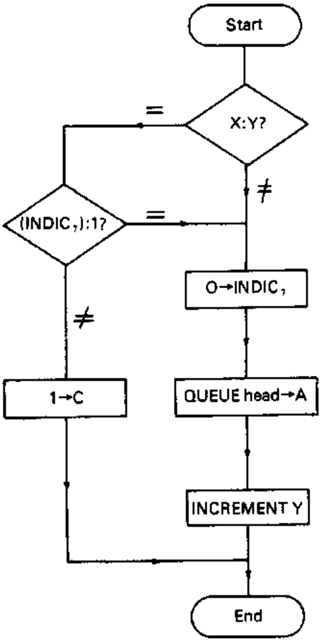


*Figure 6.7: Flow chart for withdrawing an item from a queue*

Exercise 6.4

*NB*. All programs should begin in the usual way, defining labels as necessary.

```
1.    50     LDX #0
      60     JSR OUTPUT2
      70     RTS
      80     .OUTPUT2
                .
                .
                .
                .
      150    .TEXT:]NEXT
      160    $TEXT = "This is question one"

2.    50     LDX #0
      60     JSR OUTPUT1
      70     RTS
      80     .OUTPUT1
                .
                .
```

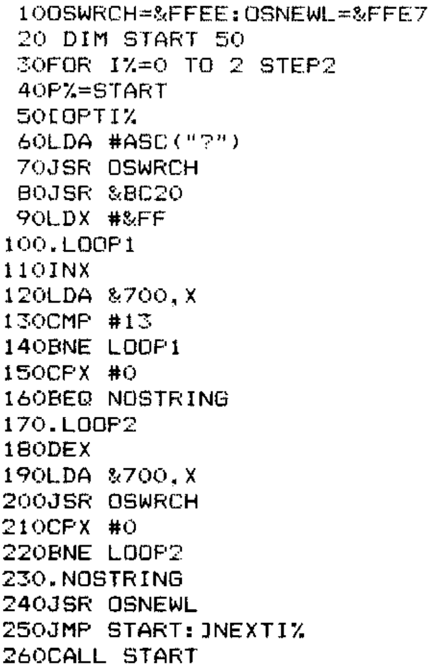
```
                 .
                 .
      160    .TEXT:]NEXT
      170    $TEXT = "This is question two"

3.    50     JSR OSNEWL
      60     RTS:]

4.    50     JSR OSNEWL
      60     LDX #0
      70     JSR OUTPUT1
      80     RTS
      90     .OUTPUT1
                 .
                 .
                 .
                 .
      170    .TEXT:]NEXT
      180    $TEXT = "This is question four"
```

Exercise 6.5

1.



```
 10OSWRCH=&FFEE:OSNEWL=&FFE7
 20 DIM START 50
 30FOR I%=0 TO 2 STEP2
 40P%=START
 50[OPTI%
 60LDA #ASC("?")
 70JSR OSWRCH
 80JSR &BC20
 90LDX #&FF
100.LOOP1
110INX
120LDA &700,X
130CMP #13
140BNE LOOP1
150CPX #0
160BEQ NOSTRING
170.LOOP2
180DEX
190LDA &700,X
200JSR OSWRCH
210CPX #0
220BNE LOOP2
230.NOSTRING
240JSR OSNEWL
250JMP START:]NEXTI%
260CALL START
```

2.

```
 10OSWRCH=&FFEE:OSASCI=&FFE3
 20 DIM START 50
 30FOR I%=0 TO 2 STEP2
 40P%=START
 50[OPTI%
 60LDA #ASC("?")
 70JSR OSWRCH
 80JSR &BC20
 90LDA &700
100CMP #ASC("*")
110BNE NTSTAR
120LDA &701
130CMP #13
140BEQ FINISH
150.NTSTAR
160LDX #&FF
170.LOOP
180INX
190LDA &700,X
200CMP #ASC(" ")
210BEQ LOOP
220JSR OSASCI
230CMP #13
240BNE LOOP
250BEQ START
260.FINISH
270 RTS:]NEXTI%
280CALL START
```

3.

```
 10OSWRCH=&FFEE
 20 DIM START 250
 30FOR I%=0 TO 2 STEP2
 40P%=START
 50[OPTI%
 60LDA #TEXT MOD 256
 70STA &37
 80LDA #TEXT DIV 256
 90STA &38
100LDA #ASC("?")
110JSR OSWRCH
120JSR &BC28
130LDX #&FF
140.LOOP1
150INX
160LDA TEXT,X
170CMP #13
180BNE LOOP1
190CPX #0
200BEQ NOSTRING
210.LOOP2
220DEX
230LDA TEXT,X
240CMP #ASC(" ")
250BEQ LOOP2
260INX
270LDA #13
280STA TEXT,X
290.NOSTRING
300RTS
310.TEXT:]NEXTI%
320CALL START
330PRINT"New length is ";LEN($TEXT)
340PRINT'"Another?"
350A$=GET$:IF A$="Y" THEN320 ELSE IF A$="N" THEN END ELSE 350
```

Exercise 6.6

(a)

```
 10DIM NUM1(3),NUM2(3),RESULT(3)
 20DIM START 50
 30P%=START
 40FOR I%=0 TO 3:NUM1(I%)=&70+I%
 50NUM2(I%)=&74+I%:RESULT(I%)=&78+I%
 60GOSUB140:NEXTI%
 70[OPT2
 80RTS:]
 90REPEAT
100INPUT"Numbers to be added",!NUM1(0),!NUM2(0)
110CALLSTART
120PRINT!NUM1(0)+!NUM2(0),!RESULT(0)
130UNTIL FALSE
140[OPT2:LDA NUM1(I%):]
150IF I%=0 THEN [OPT2:CLC:]
160[OPT2:ADC NUM2(I%)
170STA RESULT(I%):]
180RETURN
```

(b)

```
 10DIM NUM1(3),NUM2(3),RESULT(3)
 20DIM START 50
 30REPEAT
 40PRINT"Add or subtract (A/S)?";
 50REPEAT: A$=GET$
 60UNTIL A$ ="A" OR A$="S"
 70P%=START
 80FOR I%=0 TO 3:NUM1(I%)=&70+I%
 90NUM2(I%)=&74+I%:RESULT(I%)=&78+I%
100GOSUB170:NEXTI%
110[OPT2
120RTS:]
```

```
130INPUT'"Numbers",!NUM1(0),!NUM2(0)
140CALLSTART
150PRINT!NUM1(0)+(A$="S")*!NUM2(0)-(A$="A")*!NUM2(0),
   !RESULT(0)
160UNTIL FALSE
170[OPT2:LDA NUM1(I%):]
180IF I%=0 AND A$="A" THEN [OPT2:CLC:] ELSE IF I%=0 THEN
   [OPT2:SEC:]
190IF A$="A" THEN [OPT2:ADC NUM2(I%):] ELSE [OPT2:SBC
   NUM2(I%):]
200[OPT2:STA RESULT(I%):]
210RETURN
```

## Exercise 7.1

1. !NEWLOC < !OLDLOC or !NEWLOC ⩾ !OLDLOC + !NUML.

2.

```
 10CLS
 20NUM=&70:OLDLOC=&72:NEWLOC=&74
 30DIM START 100
 40FOR I%=0 TO 2 STEP 2:P%=START
 50[OPTI%
 60LDA OLDLOC
 70CLC
 80ADC NUM
 90STA OLDLOC
100LDA OLDLOC+1
110ADC NUM+1
120STA OLDLOC+1
130DEC OLDLOC+1
140LDA NEWLOC
150CLC
160ADC NUM
170STA NEWLOC
180LDA NEWLOC+1
190ADC NUM+1
200STA NEWLOC+1
210DEC NEWLOC+1
220LDY #&FF
230LDX NUM+1
240BEQ LOLOOP
250.LOOP1
260LDA (OLDLOC),Y
270STA (NEWLOC),Y
280DEY
290CPY #&FF
300BNE LOOP1
310DEC OLDLOC+1
320DEC NEWLOC+1
330DEX
340BNE LOOP1
350.LOLOOP
360LDX NUM
370BEQ FINISH
380.LOOP2
390LDA (OLDLOC),Y
400STA (NEWLOC),Y
410DEY
420DEX
430BNE LOOP2
440.FINISH
450RTS:]NEXTI%
460INPUT"How many bytes will be moved",!NUM
470INPUT"Starting address of memory to be moved",A$:
   !OLDLOC=EVAL(A$)
480INPUT"Starting address of new location",B$:!NEWLOC=EVAL(B$)
490CALL START:PRINT"Memory moved. Checking now."
500A=EVAL(A$):B=EVAL(B$)
510FOR I%=0 TO 256*?(NUM+1)+?NUM-1
520IF ?(A+I%)<>?(B+I%) PRINT "Error at move"I%+1:END
530NEXTI%
540PRINT"Check OK":GOTO460
```

This works if !NEWLOC > !OLDLOC  or !OLDLOC ≥ !NEWLOC + !NUML.

Listing 7.3 is more efficient.

3. (i) No provision for (NUM + 1) = 0. Put

```
     65     BEQ NOHIGH    and     175    .NOHIGH
```

(ii) In the move of the (NUM) residual bytes, a major fault occurs. For example, if !OLDLOC = &9460, !NEWLOC = &4000 and !NUM = &1040, when the residual bytes are to be moved !OLDLOC is &A400, !NEWLOC is &5000 and X = &40. The first move is &A440 to &5040 and the last &A401 to &5001. &A400 to &5000 has been missed out, and &A440 has been moved erroneously.

This is not correctable without reverting to the method in Listing 7.3. The fault is obscured if (NUM) = 0, or if (NUM + 1) = 0 and the correction in (i) is not made.

(iii) The routine fails if !NUM ≥ 33K or if !OLDLOC - !NEWLOC < 256. Neither of these are correctable.

### Exercise 7.2

```
10NUMBER=&70:FIRST=&72:SECOND=&74:TEMP=&76:RECLENGTH=&77:
  KEYSTART=&78:KEYEND=&79:BASE=&7A:LOOPCOUNTH=&7C
20DIM START 150
30FOR I%=0 TO 2 STEP 2:P%=START
40[OPTI%
50LDA BASE
60STA SECOND
```

```
 70LDA BASE+1
 80STA SECOND+1
 90LDX #0
100STX LOOPCOUNTH
110.BEGIN
120LDY KEYSTART
130LDA SECOND+1
140STA FIRST+1
150LDA SECOND
160STA FIRST
170CLC
180ADC RECLENGTH
190STA SECOND
200BCC LOOP1
210INC SECOND+1
220.LOOP1
230LDA (FIRST),Y
240CMP (SECOND),Y
250BCC NEWRECORD
260BNE SWAP
270INY
280CPY KEYEND
290BCC LOOP1
300BEQ LOOP1
310BCS NEWRECORD
320.SWAP
330LDY RECLENGTH
340.LOOP2
350DEY
360LDA (FIRST),Y
370STA TEMP
380LDA (SECOND),Y
390STA (FIRST),Y
400LDA TEMP
410STA (SECOND),Y
420CPY #0
430BNE LOOP2
440.NEWRECORD
450INX
460BNE NTZERO
470INC LOOPCOUNTH
480.NTZERO
490CPX NUMBER
500BNE BEGIN
510LDA LOOPCOUNTH
520CMP NUMBER+1
530BNE BEGIN
540DEC NUMBER
550BEQ LOWZERO
560LDA NUMBER
570CMP #&FF
580BNE START
590DEC NUMBER+1
600BPL START
610.LOWZERO
```

```
620LDA NUMBER+1
630BNE START
640RTS:]NEXT
650 CLS:INPUT"What is the record length",R:?RECLENGTH=R+1
660INPUT'"What are the limits for the key",?KEYSTART,?KEYEND
670INPUT'"How many records",N:!NUMBER=N-1
680DIM B ?(RECLENGTH)*N:  !BASE=B
690PRINT'"Setting up strings now"
700 FOR I%=0 TO N-1:FOR J%= 0 TO R-1:?(B+I%*(R+1)+J%)
   =RND(26)+64:NEXTJ%:?(B+I%  *(R+1)+J%)=13 :NEXTI%
710PRINT"Sorting now.":CALLSTART:PRINT"Checking."
720FOR I%=0 TO (?RECLENGTH)*(N-2) STEP (?RECLENGTH):
   IF MID$($(B+I%),?(KEYSTART)+1,?KEYEND-?KEYSTART+1)>MID$
   ($(B+I%+(?RECLENGTH)),?(KEYSTART)+1,?KEYEND-?KEYSTART+1)
   THEN PRINT "ERROR AT"STR$(I%):END
730NEXT:PRINT"O.K.":END
```

The reasons for this considerable increase in complication when dealing with more than 256 records will be discussed in Section 10.3.

Exercise 7.3

```
 10NUMBER=&70:FIRST=&72:SECOND=&74:TEMP=&76:BASE=&77:
   LOOPCOUNT=&79
 20DIM START 150
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA BASE
 60STA SECOND
 70LDA BASE+1
 80STA SECOND+1
 90LDA #0
100STA LOOPCOUNT
110STA LOOPCOUNT+1
120.BEGIN
130LDY #0
140LDA SECOND+1
150STA FIRST+1
160LDA SECOND
170STA FIRST
180CLC
190ADC #4
200STA SECOND
210BCC NOCARRY
220INC SECOND+1
230.NOCARRY
240LDX #4
250SEC
260.LOOP1
270LDA (SECOND),Y
280SBC (FIRST),Y
290INY
300DEX
310BNE LOOP1
320BVC NOOVFLOW
330EOR #&80
340.NOOVFLOW
350EOR #0
360BPL OVER
370DEY
380.LOOP2
390LDA (FIRST),Y
400STA TEMP
410LDA (SECOND),Y
420STA (FIRST),Y
430LDA TEMP
440STA (SECOND),Y
450DEY
460BPL LOOP2
470.OVER
480INC LOOPCOUNT
490BNE NTZERO
500INC LOOPCOUNT+1
510.NTZERO
520LDA LOOPCOUNT
530CMP NUMBER
540BNE BEGIN
550LDA LOOPCOUNT+1
560CMP NUMBER+1
570BNE BEGIN
580DEC NUMBER
590BEQ LOWZERO
600LDA NUMBER
610CMP #&FF
620BNE START
630DEC NUMBER+1
640BPL START
650.LOWZERO
660LDA NUMBER+1
670BNE START
680RTS:]NEXT
690CLS:INPUT"How many numbers",N:!NUMBER=N-1:DIM B 4*!NUMBER:
   !BASE=B
700FOR I%=0 TO N-1:!(B+4*I%)=RND:NEXTI%
710PRINT"Numbers assigned.  Sorting now":CALLSTART:PRINT"Done
   Checking now."
720FOR I%=0 TO N-2:IF  !(B+4*I%)>!(B+4+4*I%) THEN PRINT
   "ERROR AT "STR$(I%):END
730NEXTI% :PRINT"Checking O.K.":END
```

## Exercise 7.4

Identical to changes made in Ex.7.2, with LOOP4 and LOOP3 instead of BEGIN and START respectively. The pointer allocation in line 20 will need to be increased also.

Exercise 7.5

```
1.          CMP #&61
            BCC OVER
            CMP #&7B
            BCS OVER
            SEC
            SBC #&20
       OVER JMP &E1BB

2.          LDA #ASC("?")
            JSR OSWRCH
            JMP &DFA5
```

Exercise 8.1

```
1. Box  1    becomes   0 → RES
                       0 → RES+1
                       0 → RES+2
                       0 → RES+3

   Box  2    becomes   Shift (MULTER+1; MULTER) right

   Box  3    becomes   (RES+3; RES+2) + (MULTED+1; MULTED)
                                → RES+3; RES+2

   Box  4    becomes   Rotate (RES+3; RES+2; RES+1; RES) right
```

```
 10MULTER=&70:MULTED=&72:RES=&74
 20DIM START 50
 30FOR I%=0 TO 2 STEP2:P%=START
 40[OPTI%
 50LDA #0
 60STA RES
 70STA RES+1
 80STA RES+2
 90STA RES+3
100LDX #16
110.LOOP
120LSR MULTER+1
130ROR MULTER
140BCC ZERO
150LDA RES+2
160CLC
170ADC MULTED
180STA RES+2
190LDA RES+3
200ADC MULTED+1
210STA RES+3
220.ZERO
230ROR RES+3
240ROR RES+2
250ROR RES+1
260ROR RES
270DEX
280BNE LOOP
290RTS:]NEXTI%
300CLS:REPEAT
310 INPUT"Numbers to be multiplied",A,B:!MULTER=A:!MULTED=B
320CALLSTART
330PRINTA*B,16777216*?(RES+3)+65536*?(RES+2)+256*?
   (RES+1)+?RES
340UNTIL FALSE
```

The accumulator is used for the multiple precision add in lines 160 to 210 so it cannot be used to store part of the result.

2.

```
 10MULTER=&70:MULTED=&71:RES=&72
 20DIM START 150
 30FOR I%=0 TO 2 STEP2: P%=START
 40[OPTI%
 50LDY #0
 60LDX #0
 70LDA MULTER
 80BPL PLUS1
 90INX
100EOR #&FF
110CLC
120ADC #1
130STA MULTER
140.PLUS1
150LDA MULTED
160BPL PLUS2
170DEX
180EOR #&FF
190CLC
200ADC #1
210STA MULTED
220.PLUS2
230TXA
240BEQ PLUS
250LDY #1
260.PLUS
270LDA #0
280STA RES
290LDX #8
300.LOOP
310LSR MULTER
320BCC ZERO
330CLC
340ADC MULTED
350.ZERO
360ROR A
370ROR RES
380DEX
390BNE LOOP
400STA RES+1
410TYA
420BEQ ANSPL
430SEC
440LDA #0
450SBC RES
460STA RES
470LDA #0
480SBC RES+1
490STA RES+1
500.ANSPL
510RTS:]NEXTI%
520CLS:REPEAT
530INPUT"Numbers to be multiplied",A,B
540 IF A<0 THEN C=256+A ELSE C=A
550 IF B<0 THEN D=256+B ELSE D=B
560 ?MULTER=C:?MULTED=D:CALLSTART
570PRINTA*B,
580IF ?(RES+1)>127 THEN PRINT(?(RES+1)-255)*256+?RES-256
   ELSE PRINT?(RES+1)*256+?RES
590UNTIL FALSE
```

The details are:

50 Flag for sign of result.

60 Indicator for each sign.

70-130 Check if multiplier negative, and if so increment X and obtain two's complement of multiplier.

150-210 Check if multiplied is negative, and if so decrement X and obtain two's complement of multiplied.

230-250 If X is not zero, result will be negative. Set flag in Y.

270-400 As Listing 8.3.

410-490 If result is to be negative, form two's complement of result by subtracting from zero.

3. There are many possible solutions, one of which is to use Listing 8.3, but the following is shorter and quicker. It does the computation by evaluating 256 * Y + X - 6 * Y. It is often the case that knowledge of the multiplier can allow quicker routines than the general-purpose Listing 8.3, and this is such a case.

```
 10RES=&70
 20DIM START 50
 30FOR I%=0 TO 2 STEP 2
 40P%=START
 50[OPTI%
 60STY MEMLOC+1
 70STY RES+1
 80TXA
 90LDY #6
100.LOOP
110SEC
120.MEMLOC
130SBC #0 Dummy operand
140BCS NOBORROW
150DEC RES+1
160.NOBORROW
170DEY
180BNE LOOP
190STA RES
200LDY MEMLOC+1
210RTS:]NEXTI%
220CLS:REPEAT
230INPUT"What are X and Y",X%,Y%
240CALLSTART
250PRINT250*Y%+X%,256*?(RES+1)+?RES
260UNTIL FALSE
```

Lines 70 and 80 compute 256 * Y + X (with X in the accumulator). Lines 90 to 180 subtract Y six times, giving 250 * Y + X.

It is possible to use TEMP to save Y, instead of the internal location MEMLOC, but whilst this saves 2 bytes it costs 4 cycles of time (see Appendix 1).

## Exercise 8.2

```
Put     193    DEC NUMBER+1
        196    BPL BACK
```

and change ?NUMBER to !NUMBER in line 400.

## Exercise 8.3

1. The REPWHILE loop executes infinitely. Put BEQ MISTAKE at 75, where MISTAKE is some error-handing routine (see Listing 8.8).

2.

```
10DVID=&70:DVIS=&72:QUOT=&74
20DIM START 50
30FOR I%=0 TO 2 STEP 2:P%=START
```

```
 40[OPTI%
 50LDX #0
 60LDA #0
 70STA QUOT
 80STA QUOT+1
 90LDA DVIS+1
100BMI LOOP
110ORA DVIS
120BEQ FINISH
130.REPWHILE
140INX
150ASL DVIS
160ROL DVIS+1
170BPL REPWHILE
180.LOOP
190LDA DVID
200CMP DVIS
210LDA DVID+1
220SBC DVIS+1
230BCC LESS
240INC QUOT
250LDA DVID
260SEC
270SBC DVIS
280STA DVID
290LDA DVID+1
300SBC DVIS+1
310STA DVID+1
320.LESS
330DEX
340BMI FINISH
350LSR DVIS+1
360ROR DVIS
370ASL QUOT
380ROL QUOT+1
390JMP LOOP
400.FINISH
410 RTS:]NEXTI%
420CLS:REPEAT
430INPUT"Dividend",DD
440INPUT"Divisor",DS
450!DVID=DD:!DVIS=DS
460CALLSTART
470PRINTDD DIV DS,DD MOD DS
480PRINT256*?(QUOT+1)+?QUOT,256*?(DVID+1)+?DVID
490UNTIL FALSE
```

3. The following code should be inserted after line 390 in the listing in Q.2, replacing BEQ or BMI FINISH in lines 120 and 340 by BMI ROUND. Alter lines 470 and 480 accordingly.

```
ROUND  LSR DVIS+1
       ROR DVIS
       BCC NOFRAC      Divide divisor by two.
       INC DVIS        If there is a half,
       BNE NOFRAC      then round up.
       INC DVIS+1
NOFRAC LDA DVID
       CMP DVIS        Compare remainder (in DVID)
       LDA DVID+1      with half the divisor (in DVIS).
       SBC DVIS+1
       BCC FINISH
       INC QUOT        If remainder is at least half
       BNE FINISH      the divisor round up.
       INC QUOT+1
FINISH RTS
```

4.

```
 10DVID=&70:DVIS=&71:QUOT=&72
 20DIM START 150
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDY #0
 60LDX #0
 70LDA DVIS
 80BPL PLUS1
 90INX
100EOR #&FF
110CLC
120ADC #1
130STA DVIS
140.PLUS1
150LDA DVID
160BPL PLUS2
170DEX
180EOR #&FF
190CLC
200ADC #1
210STA DVID
220.PLUS2
230TXA
240BEQ PLUS
250LDY #1
260.PLUS
270LDX #0
280STX QUOT
290LDA DVIS
300BEQ ANSPL
310.REPWHILE
320BMI LOOP
330INX
340ASL DVIS
350BPL REPWHILE
360.LOOP
370LDA DVID
380CMP DVIS
390BCC LESS
400INC QUOT
410LDA DVID
420SEC
430SBC DVIS
440STA DVID
450.LESS
460DEX
470BMI ROUND
480LSR DVIS
490ASL QUOT
500JMP LOOP
510.ROUND
520LSR DVIS
530BCC NOFRAC
540INC DVIS
550.NOFRAC
560LDA DVID
570CMP DVIS
580BCC SIGN
590INC QUOT
600.SIGN
610TYA
620BEQ ANSPL
630SEC
640LDA #0
650SBC QUOT
660STA QUOT
670.ANSPL
680RTS:]NEXTI%
690CLS:REPEAT
700INPUT"Dividend",A:IF A<0 THEN C=256+A ELSE C=A
710INPUT"Divisor",B:IF B<0 THEN D=256+B ELSE D=B
720?DVID=C:?DVIS=D:CALLSTART
730PRINTA/B
740IF ?QUOT>127 PRINT ?QUOT-256 ELSE PRINT?QUOT
750UNTIL FALSE
```

The details are:

50-250 Virtually identical to first half of solution for Q.2, Ex.8.1.

270-500 Identical to Listing 8.7 (with the addition in Q.1).

520-590 Equivalent to Q.3 for 8-bit numbers.

610-660 Identical to last part of Q.2, Ex.8.1.

## Exercise 8.4

```
 10DVID=&70:DVIS=&72:QUOTH=&74:OSWRCH=&FFEE
 20DIM START 100
 30FOR I%=0 TO 2 STEP 2:P%=START
 40[OPTI%
 50LDA DVIS
 60ORA DVIS+1
 70BEQ MISTAKE
 80LDA #0
 90STA QUOTH
100LDX #16
110.LOOP
120ASL DVID
130ROL DVID+1
140ROL A
150TAY
160ROL QUOTH
170CMP DVIS
180LDA QUOTH
190SBC DVIS+1
200BCC LESS
210TYA
220SBC DVIS
230TAY
240LDA QUOTH
250SBC DVIS+1
260STA QUOTH
270INC DVID
280.LESS
290TYA
300DEX
310BNE LOOP
320RTS
330.MISTAKE
340LDA #ASC("?")
350JSR OSWRCH
360LDA #7
370JSR OSWRCH
380RTS:]NEXTI%
390CLS:REPEAT
400INPUT"Dividend",A
410INPUT"Divisor",B
420!DVID=A:!DVIS=B
430!&403=USRSTART
440PRINTA DIV B,A MOD B
450PRINT256*?(DVID+1)+?DVID,256*?QUOTH+?&403
460UNTIL FALSE
```

The details are:

50-70 Output error message if (DVIS+1; DVIS) is zero.

120-160 Shift left (DVIS+1; DVIS) one bit into (QUOTH; A) saving the accumulator temporarily in Y.

170-200 Compare (QUOTH; A) to (DVIS+1; DVIS).

210-270 If quotient not less than divisor, retrieve the accumulator, subtract (DVIS) (the carry must already be set), save the accumulator in Y, subtract (DVIS+1) from (QUOTH) with any borrow, and increment the dividend by one.

290-320 Retrieve the accumulator, loop 16 times, and then return.

340-380 If a division by zero occurs, output a query sign and a short beep, and return.

The program is shorter and quicker.

Exercise 9.1

1.

```
        PHP
        PLA
        ORA #&C0
        PHA
        PLP
```

2.

```
        SEC
        PHP
  LOOP1 PLP
        LDA (SECOND), Y
        SBC (FIRST), Y
        INY
        PHP
        CPY #4
        BNE LOOP1
```

The first is more efficient in memory space, since fewer bytes are used; and more efficient in time in that the loop here contains 2 excess instructions.

Exercise 9.2

```
1.          TAY
            TXA
            PHA
            TYA
            TAX
            PLA


2. (i)      PHA
            STX MEMLOC+1
            SEC
     MEMLOC SBC #0            Dummy
            TAX
            PLA

   (ii)     PHA
            TXA
            STY MEMLOC+1
            CLC
     MEMLOC ADC #0            Dummy
            TAX
            PLA
```

Exercise 9.3

```
                   Code             Time
                   SEI               2  }
                   PHP               3  }
                   PHA               3  }
                   TXA               2  }  15
                   PHA               3  }
                   LDX #t            2  }
             LOOP1 LDA #c            2
                   STA DELAY         3
             LOOP2 DEC DELAY         5
                   BNE LOOP2         3
```

$$\begin{array}{ll} \textcircled{A} & \\ \text{DEX} & 2 \\ \text{BNE LOOP1} & 3 \\ \textcircled{B} & \end{array}$$

$$\left.\begin{array}{ll} \text{PLA} & 4 \\ \text{TAX} & 2 \\ \text{PLA} & 4 \\ \text{PLP} & 4 \end{array}\right\} 14$$

$$15 + 14 + 10t - 1 + (8c - 1) * t = 2000t \qquad \textcircled{1}$$

$$\Rightarrow c = \frac{1991t - 28}{8t}$$

When t = 10, c = 248; and left hand side of equation $\textcircled{1}$ is 19958. Hence we require 42 cycles.

Put NOP : NOP at $\textcircled{A}$ and NOP at $\textcircled{B}$ .

## Exercise 9.4

```
10COLUMNS=&70:ROWS=&71:COLCOPY=&72:LIMIT=&73:BEGINCONTROL=
  &74:HIBYTE=&75:LOCAION=&76:STORE=&78:OSWRCH=&FFEE:OSBYTE=
  &FFF4
20FORI%=0 TO 2 STEP 2:P%=&D00:RESTORE
30[OPTI%
40LDA #3
50STA LIMIT
60LDX #0
70JSR CONTROL
80LDX #4
90LDA #&85
100JSR OSBYTE
110STY MEMLOC+1
120LDA #&84
130JSR OSBYTE
140STY HIBYTE
150.MEMLOC
160CPY #0 Dummy operand
170BNE ZEROMODE
180LDA #7
190STA LIMIT
200LDA #3
210STA BEGINCONTROL
220LDA #40
230BNE FOURMODE
240.ZEROMODE
250LDA #11
```

```
260STA LIMIT
270LDA #7
280STA BEGINCONTROL
290LDA #80
300.FOURMODE
310STA COLUMNS
320LDA &322
330STA LOCATION
340LDA &323
350STA LOCATION+1
360LDA #32
370STA ROWS
380.BEGIN
390LDA COLUMNS
400STA COLCOPY
410LDX BEGINCONTROL
420JSR CONTROL
430.LOOP1
440LDY #7
450.LOOP2
460LDA (LOCATION),Y
470STA STORE,Y
480DEY
490BPL LOOP2
500LDY #8
510.LOOP3
520LDX #7
530LDA #1
540JSR OSWRCH
550.LOOP4
560ASL STORE,X
570ROR A
580DEX
590BPL LOOP4
600JSR OSWRCH
610DEY
620BNE LOOP3
630LDA LOCATION
640CLC
650ADC #8
660STA LOCATION
670BCC NOCARRY
680INC LOCATION+1
690.NOCARRY
700DEC COLCOPY
710BNE LOOP1
720LDA #1
730JSR OSWRCH
740LDA #&0D
750JSR OSWRCH
760LDA LOCATION+1
770BPL OVER
780LDA HIBYTE
790STA LOCATION+1
800.OVER
```

```
810DEC ROWS
820BNE BEGIN
830LDA #13
840STA LIMIT
850LDX #11
860JSR CONTROL
870RTS
880.CONTROL
890LDA #1
900JSR OSWRCH
910LDA TABLE,X
920JSR OSWRCH
930INX
940CPX LIMIT
950BNE CONTROL
960RTS
970.TABLE:]NEXTI%
980FOR I%=1 TO 13
990READ ?P%
1000P%=P%+1:NEXTI%
1010DATA27,65,8,27,75,64,1,27,76,128,2,27,50
```

The details are:

40-130 As Listing 8.6.

140 Only the high byte of physical screen memory is required to be saved.

150-310 As Listing 8.6, 160-320.

320-350 Store start of actual screen memory (i.e. taking into account changes due to scrolling) in LOCATION. All addresses will be as a model B, even in a model A machine. The OS takes care of this.

360-750 As Listing 8.6, 330-720.

760-790 If we go beyond &7FFF, the high byte becomes negative, and we replace it by the high byte of the start of physical screen memory (i.e. the start of actual screen memory when there has been no scrolling).

800-1010 As Listing 8.6, 730-930.

Notice that this program is longer, and so if we know that there will be no scrolling (typically so when using the high-resolution graphics), Listing 8.6 is the better choice if memory is at a premium.

# *Appendix 1 6502 Instruction Set*

It is convenient to divide the 56 instructions up into four groups, depending upon how many of the bits in their op-codes are fixed.

## GROUP 1: FIVE BITS FIXED

Instructions in this group have fixed (f) and variable (v) bits as follows:

fffvvvff

There are two subgroups to consider:

### GROUP 1A: 8 ADDRESSING MODES

These modes are:

| *vvv* | *Mode* |
|---|---|
| 000 | Indexed indirect (see Appendix 3) |
| 001 | Zero page |
| 010 | Immediate (not STA) |
| 011 | Absolute |
| 100 | Indirect indexed |
| 101 | Zero page, indexed X |
| 110 | Absolute, indexed Y |
| 111 | Absolute, indexed X |

The instructions in this group are:

ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

## GROUP 1B: 5 ADDRESSING MODES

The modes are:

| *vvv* | *Mode* |
|---|---|
| 000 | Immediate (not ASL, LSR, ROL, ROR) |
| 001 | Zero page |
| 010 | Accumulator (not LDX, LDY) |
| 011 | Absolute |
| 101 | Zero page, indexed X (indexed Y in LDX) |
| 111 | Absolute, indexed X (indexed Y in LDX) |

The instructions in this group are:

ASL, LDX, LDY, LSR, ROL, ROR

## GROUP 2: SIX BITS FIXED

There are two subgroups:

## GROUP 2A

These have fixed (f) and variable (v) bits as follows:

fffvvfff

The addressing modes are:

| *vv* | *Mode* |
|---|---|
| 00 | Zero page |
| 01 | Absolute |
| 10 | Zero page, indexed X (indexed Y in STX) |
| 11 | Absolute, indexed X (not STX, STY) |

The instructions in this group are:

DEC, INC, STX, STY

## GROUP 2B

These have fixed (f) and variable (v) bits as follows:

| *vv* | *Mode* |
|---|---|
| 00 | Immediate |

| | |
|---|---|
| 01 | Zero page |
| 11 | Absolute |

The instructions in this group are:

CPX, CPY

## GROUP 3: SEVEN BITS FIXED

There are two subgroups:

### GROUP 3A

This has a fixed (f) and variable (v) bit pattern of

ffffvfff

The modes are:

| *v* | *Mode* |
|---|---|
| 0 | Zero page |
| 1 | Absolute |

The only instruction in this group is BIT.

### GROUP 3B

This has a fixed (f) and variable (v) bit pattern of

ffvfffff

The modes are:

| *v* | *Mode* |
|---|---|
| 0 | Absolute |
| 1 | Indirect |

The only instruction in this group is JMP.

## GROUP 4: ALL BITS FIXED

These are the implied and relative addressing mode instructions:

BCC, BCS, BEQ, BMI, BNE, BPL, BRK, BVC, BVS, CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, PHA, PHP, PLA, PLP, RTI, RTS, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS, TYA. Also the absolute mode for JSR.

As an example of this, consider LDA. This in in group 1A, and the fixed bits are 101vvv01. Taking each set of values for vvv in turn we arrive at the opcodes A1, A5, A9, AD, B1, B5, B9 and BD.

In the detailed summary which follows the fixed bytes will be given for each mnemonic, and then each addressing mode will have attached its own variable bits. This information is useful if one wishes to construct an assembler or disassembler, for example. (The disassembler on Tape #1, available with this book, uses this information to speed up disassembly.)

ABBREVIATIONS IN TABLE A1.1

| | |
|---|---|
| * | Plus 1 cycle if page boundary crossed |
| + | Plus 1 cycle if branch occurs; plus 2 cycles if branch crosses into another page |
| n | Number of bytes comprising the op-code and operand |
| t | Number of machine cycles needed to complete instruction |
| v | A variable bit in the opcode |
| M | An arbitrary memory location (i.e. an address) |
| (M) | The contents of M. |
| $M_6$ | The contents of bit 6 of M. |
| $\overline{M}$ | The one's complement of (M) |
| r | A signed byte (i.e. &00 to &7F is +0 to +127; &80 to &FF is -128 to -1). |
| LOOP | An arbitrary label (i.e. an address) |
| N | The negative flag |
| Z | The zero flag |
| C | The carry flag |
| V | The overflow flag |
| I | The interrupt disable flag |
| D | The decimal mode flag |
| B | The break flag |
| A | The accumulator |
| X,Y | The index registers |
| P | The processor status register |
| S | The stack pointer |
| PC | Program counter (containing the address of the *first byte of the instruction*) |
| → | Copy to memory location or register |
| ↑ | Copy to stack (i.e. push) |
| ↓ | Transfer from stack (i.e. pull) |
| $\vee$ | OR |
| $\wedge$ | AND |
| $\veebar$ | Exclusive-OR |
| $\oplus$ | Signed addition (i.e. second byte is treated as a signed byte) |
| ✓ | Flag is affected by instruction |
| — | Flag is not affected by instruction |

*Table A1.1: Alphabetical summary of instruction set*

## ADC

**Description of ADC M**

Add the contents of M to the accumulator, together with any carry bit, store the result in the Accumulator and any carry in the carry flag.

**Symbolic operation of ADC M**

A + (M) + C → A

Carry → C

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

**Fixed bit pattern**

011vvv01

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | 69 | 2 | 65 | 2 | 6D | 3 | 75 | 2 | 7D | 3 | 79 | 3 | 71 | 2 | | | 61 | 2 |
| t | vvv | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## AND

**Description of AND M**

Perform the logical AND operation bit by bit on the corresponding bits of (M) and the accumulator, leaving the results in the accumulator

**Symbolic operation of AND M**

A ∧ (M) → A

| ∧ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

**Fixed bit pattern**

001vvv01

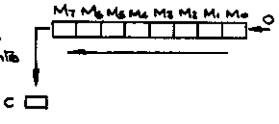
| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | 29 | 2 | 25 | 2 | 2D | 3 | 35 | 2 | 3D | 3 | 39 | 3 | 31 | 2 | | | 21 | 2 |
| t | vvv | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## ASL

**Description of ASL M**

Move the contents of M left one bit : bit 7 goes into carry. Zero goes into bit 0.
The result is in M.

**Symbolic operation of ASL M**



**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | – |

**Fixed bit pattern**

000vvv10

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 06 | 2 | 0E | 3 | 16 | 2 | 1E | 3 | | | | | 0A | 1 | | |
| t | vvv | | | 5 | 001 | 6 | 011 | 6 | 101 | 7 | 111 | | | | | 2 | 010 | | |

## BCC

| Description of<br>Bcc Loop | Symbolic operation of<br>Bcc r | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| If C=0, branch to the instruction labelled Loop | If C=0:<br>PC+2 ⊕ r → PC<br>If C=1:<br>no operation | Relative | 90 | 2 | $2^{\dagger}$ |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| — | — | — | — |

## BCS

| Description of<br>BCS Loop | Symbolic operation of<br>BCS r | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| If C=1, branch to the instruction labelled Loop | If C=1:<br>PC+2 ⊕ r → PC<br>If C=0<br>no operation | Relative | B0 | 2 | $2^{\dagger}$ |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| — | — | — | — |

## BEQ

| Description of<br>BEQ Loop | Symbolic operation of<br>BEQ r | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| If Z=1, branch to the instruction labelled Loop | If Z=1:<br>PC+2 ⊕ r → PC<br>If Z=0<br>no operation | Relative | F0 | 2 | $2^{\dagger}$ |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| — | — | — | — |

## BIT

**Description of BIT M**

The logical AND of M and A is performed, the result not being Stored. Z is set to 1 if the result is Zero, otherwise Z is Zero. Finally bits 6 and 7 of M are copied to V and N. A is unchanged.

**Symbolic operation of BIT M**

$\overline{A \wedge (M)} \rightarrow Z$

$M_6 \rightarrow V$

$M_7 \rightarrow N$

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| $M_7$ | ✓ | — | $M_6$ |

**Fixed bit pattern**

00I0VI00

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 24 | 2 | 2C | 3 | | | | | | | | | | | | |
| t | v | | | 3 | 0 | 4 | 1 | | | | | | | | | | | | |

## BMI

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| BMI LOOP | BMI r | | | | | N | Z | C | V |
| If N=1, branch to the instruction labelled LOOP | If N = 1:<br>Pc+2⊕r → PC<br>If N = 0:<br>no operation | Relative | 30 | 2 | $2^{\dagger}$ | – | – | – | – |

## BNE

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| BNE LOOP | BNE r | | | | | N | Z | C | V |
| If Z = 0, branch to the instruction labelled LOOP | If Z=0:<br>Pc+2⊕r → Pc<br>If Z = 1:<br>no operation | Relative | D0 | 2 | $2^{\dagger}$ | – | – | – | – |

## BPL

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| BPL LOOP | BPL r | | | | | N | Z | C | V |
| If N=0, branch to the instruction labelled LOOP | If N = 0:<br>PC+2⊕r → PC<br>If N = 1:<br>no operation | Relative | 10 | 2 | $2^{\dagger}$ | – | – | – | – |

## BRK

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| BRK | BRK | | | | | N | Z | C | V |
| A software interrupt. (Has a special use in the BBC microcomputer. Best not used in assembly programs.) | ↑ PC + 2<br>S−2 → S<br>↑ P<br>S−1 → S<br>(FFFF ; FFFE) → PC | Implied | 00 | 1 | 7 | – | – | – | – |

B is set to 1 before P is pushed onto the stack.

I is set to 1 after P is pushed onto the stack.

## BVC

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| BVC LOOP | BVC r | | | | | N | Z | C | V |
| If V = 0, branch to the instruction labelled LOOP | If V = 0:<br>PC + 2 ⊕ r → PC<br>If V = 1:<br>no operation | Relative | 50 | 2 | $2^t$ | – | – | – | – |

## BVS

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| BVS LOOP | BVS r | | | | | N | Z | C | V |
| If V = 1, branch to the instruction labelled LOOP | If V = 1:<br>PC + 2 ⊕ r → PC<br>If V = 0:<br>no operation | Relative | 70 | 2 | $2^t$ | – | – | – | – |

## CLC

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| CLC | CLC | | | | | N | Z | C | V |
| The carry flag is set to zero (cleared) | 0 → C | Implied | 18 | 1 | 2 | – | – | 0 | – |

## CLD

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| CLD | CLD | | | | | N | Z | C | V |
| The decimal flag is set to 0 (cleared). All arithmetic will now be standard binary | 0 → D | Implied | D8 | 1 | 2 | – | – | – | – |

D is set to Zero

## CLI

| Description of CLI | Symbolic operation of CLI | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The interrupt mask bit is set to zero, so enabling interupts. | 0 → I | Implied | 58 | 1 | 2 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

I is set to 0

## CLV

| Description of CLV | Symbolic operation of CLV | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The overflow flag is set to zero (cleared) | 0 → V | Implied | B8 | 1 | 2 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | 0 |

## CMP

**Description of CMP M**

(M) is subtracted from A, but the result is not stored and A is unchanged. Z is 1 on equality, C is 1 if (M) does not exceed A, N is 1 if bit 7 of the result is 1.

**Symbolic operation of CMP M**

$A - (M) \rightarrow$ result (not stored)

$$\left.\begin{array}{l}\text{if } A < (M), Z=0, C=0 \\ \text{if } A = (M), Z=1, C=1 \\ \text{if } A > (M), Z=0, C=1\end{array}\right\} N = \text{result}_7$$

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | – |

**Fixed bit pattern**

110VVV01

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | C9 | 2 | C5 | 2 | CD | 3 | D5 | 2 | DD | 3 | D9 | 3 | D1 | 2 | | | C1 | 2 |
| t | VVV | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## CPX

**Description of CPX M**

(M) is subtracted from X, but the result is not stored and X is unchanged. Z is 1 on equality, C is 1 if (M) does not exceed X, N is 1 if bit 7 of the result is 1.

**Symbolic operation of CPX M**

$X - (M) \rightarrow$ result (not stored)

$$\left.\begin{array}{l}\text{if } X < (M), Z=0, C=0 \\ \text{if } X = (M), Z=1, C=1 \\ \text{if } X > (M), Z=0, C=1\end{array}\right\} N = \text{result}_7$$

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | – |

**Fixed bit pattern**

1110VV00

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | E0 | 2 | E4 | 2 | EC | 3 | | | | | | | | | | | | |
| t | VV | 2 | 00 | 3 | 01 | 4 | 11 | | | | | | | | | | | | |

## CPY

| Description of CPY M | Symbolic operation of CPY M | Flags affected | Fixed bit pattern |
|---|---|---|---|
| (M) is subtracted from Y but the result is not stored and Y is unchanged. Z is 1 on equality, C is 1 if (M) does not exceed X. N is 1 if bit 7 of the result is 1. | Y-(M) → result (not stored)<br>if Y<(M), Z=0, C=0<br>if Y=(M), Z=1, C=1<br>if Y>(M), Z=0, C=1<br>N = result$_7$ | N Z C V<br>✓ ✓ ✓ — | 1100VV00 |

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | — |

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | C0 | 2 | C4 | 2 | CC | 3 | | | | | | | | | | | | |
| t | vv | 2 | 00 | 3 | 01 | 4 | 11 | | | | | | | | | | | | |

## DEC

| Description of DEC M | Symbolic operation of DEC M | Flags affected | Fixed bit pattern |
|---|---|---|---|
| The contents of M is decreased by 1. (If (M) is zero it becomes &FF.) The result is stored in M. | (M)-1 → M | N Z C V<br>✓ ✓ — — | 110VV110 |

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | — | — |

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | C6 | 2 | CE | 3 | D6 | 2 | DE | 3 | | | | | | | | |
| t | vv | | | 5 | 00 | 6 | 01 | 6 | 10 | 7 | 11 | | | | | | | | |

## DEX

| Description of DEX | Symbolic operation of DEX | Addressing mode | Opcode | n | t | Flags affected |
|---|---|---|---|---|---|---|
| The contents of X is reduced by 1. (If X is zero it becomes &FF) | X - 1 → X | Implied | CA | 1 | 2 | N Z C V<br>✓ ✓ — — |

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | — | — |

## DEY

| Description of DEY | Symbolic operation of DEY | Addressing mode | Opcode | n | t | Flags affected |
|---|---|---|---|---|---|---|
| The contents of Y is reduced by 1. (If Y is zero it becomes &FF) | Y - 1 → Y | Implied | 88 | 1 | 2 | N Z C V<br>✓ ✓ — — |

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | — | — |

## EOR

**Description of EOR M**

Perform the exclusive - OR operation bit by bit on the corresponding bits of (M) and the accumulator, leaving the result in the accumulator.

**Symbolic operation of EOR M**

$A \not\wedge (M) \rightarrow A$

| $\not\wedge$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

**Fixed bit pattern**

010vvv01

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | 49 | 2 | 45 | 2 | 4D | 3 | 56 | 2 | 5D | 3 | 59 | 3 | 51 | 3 | | | 41 | 2 |
| t | vvv | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## INC

**Description of INC M**

The contents of M is increased by 1. (If (M) is & FF it becomes Zero.)

The result is Stored in M.

**Symbolic operation of INC M**

$(M) + 1 \rightarrow M$

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

**Fixed bit pattern**

111vv110

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | E6 | 2 | EE | 3 | F6 | 2 | FE | 3 | | | | | | | | |
| t | vv | | | 5 | 00 | 6 | 01 | 6 | 10 | 7 | 11 | | | | | | | | |

## INX

| Description of INX | Symbolic operation of INX | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The contents of X is increased by 1. (If X is & FF it becomes Zero). | $X + 1 \rightarrow X$ | Implied | E8 | 1 | 2 |

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

## INY

| Description of INY | Symbolic operation of INY | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The contents of Y is increased by 1. (If Y is & FF it becomes Zero). | $Y + 1 \rightarrow Y$ | Implied | C8 | 1 | 2 |

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

## JMP

Description of JMP Loop

The addren represented by the label Loop is loaded into the program counter causing a jump to occur to the instruction at that address.

Symbolic operation of JMP Loop

Loop → PC

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

Fixed bit pattern

01V01100

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | | Indirect | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | | | 4C | 3 | | | | | | | | | | | | | 6C | 3 |
| t | v | | | | | 3 | 0 | | | | | | | | | | | | | 5 | 1 |

## JSR

| Description of JSR Loop | Symbolic operation of JSR Loop | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The program Counter plus 2 (the address minus one of the instruction following the JSR) is saved on the Stack. The address represented by Loop is loaded into the program Counter. | PC + 2 ↓<br>Loop → PC | Absolute | 20 | 3 | 6 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

## LDA

Description of LDA M

The contents of M is copied into the accumulator.

Symbolic operation of LDA M

(M) → A

Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

Fixed bit pattern

101VVV01

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | A9 | 2 | A5 | 2 | AD | 3 | B5 | 2 | BD | 3 | B9 | 3 | B1 | 2 | | | A1 | 2 |
| t | vvv | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## LDX

**Description of LDX M**

The contents of M is copied into X.

**Symbolic operation of LDX M**

(M) → X

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | — | — |

**Fixed bit pattern**

101vvv10

| | | Immediate | | Zero Page | | Absolute | | Zero page Y | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | A2 | 2 | A6 | 2 | AE | 3 | B6 | 2 | | | BE | 3 | | | | | | |
| t | vvv | 2 | 000 | 3 | 001 | 4 | 011 | 4 | 101 | | | 4* | 111 | | | | | | |

## LDY

**Description of LDY M**

The contents of M is copied into Y.

**Symbolic operation of LDY M**

(M) → Y

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | — | — |

**Fixed bit pattern**

101vvv00

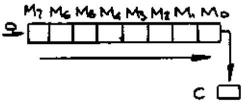
| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (indirect) Y | | Accumulator | | (indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | A0 | 2 | A4 | 2 | AC | 3 | B4 | 2 | BC | 3 | | | | | | | | |
| t | vvv | 2 | 000 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | | | | | | | | |

## LSR

**Description of LSR M**

Move the contents of M right one bit: bit 0 goes into carry. Zero goes into bit 7. The result is in M.

**Symbolic operation of LSR M**



**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| 0 | ✓ | ✓ | — |

**Fixed bit pattern**

010vvv10

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 46 | 2 | 4E | 3 | 56 | 2 | 5E | 3 | | | | | 4A | 1 | | |
| t | vvv | | | 5 | 001 | 6 | 011 | 6 | 101 | 7 | 111 | | | | | 2 | 010 | | |

## NoP

| Description of NoP | Symbolic operation of NoP | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| Does Nothing for 2 cycles. | — | Implied | EA | 1 | 2 |

**Flags affected**

| N | Z | C | V |
|---|---|---|---|
| — | — | — | — |

## ORA.

Description of ORA M

Perform the inclusive -OR operation bit by bit on the corresponding bits of (M) and the accumulator, leaving the result in the accumulator.

Symbolic operation of ORA M

$A \vee (M) \rightarrow A$

| V | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

Fixed bit pattern

000VVV01

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | 09 | 2 | 05 | 2 | 0D | 3 | 15 | 2 | 1D | 3 | 19 | 3 | 11 | 2 | | | 01 | 2 |
| t | VVV | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## PHA

| Description of PHA | Symbolic operation of PHA | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The contents of the accumulator is copied to the stack, and the stack pointer is decreased by one. | ↑A<br>S−1 → S | Implied | 48 | 1 | 3 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

## PHP

| Description of PHP | Symbolic operation of PHP | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The contents of the status register is copied to the stack, and the stack pointer is decreased by 1. | ↑P<br>S−1 → S | Implied | 08 | 1 | 3 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

## PLA

| Description of PLA | Symbolic operation of PLA | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The contents of the accumulator is filled by the last byte pushed on the stack, and the stack pointer is increased by one. | ↓A<br>S+1 → S | Implied | 68 | 1 | 4 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

## PLP

Description of PLP

The status register is filled with the last byte pushed onto the stack, and the stack pointer is increased by one.

Symbolic operation of PLP

↓ P
S+1 → S

| Addressing mode | Opcode | n | t |
|---|---|---|---|
| Implied | 28 | 1 | 4 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

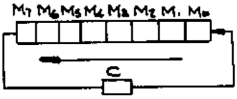
B.D and I are also affected.

## ROL

Description of ROL M

Move the contents of M left one bit: bit 7 goes into carry after the present contents of carry has gone into bit 0. The result is in M.

Symbolic operation of ROL M



Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | — |

Fixed bit pattern

001VVV10

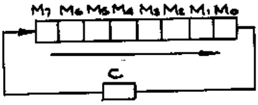
| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 26 | 2 | 2E | 3 | 36 | 2 | 3E | 3 | | | | | 2A | 1 | | |
| t | vvv | | | 5 | 001 | 6 | 011 | 6 | 101 | 7 | 111 | | | | | 2 | 010 | | |

## ROR

Description of ROR M

Move the contents of M right one bit: bit 0 goes into carry after the present contents of carry has gone into bit 7. The result is in M.

Symbolic operation of ROR M



Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | — |

Fixed bit pattern

011VVV10

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 66 | 2 | 6E | 3 | 76 | 2 | 7E | 3 | | | | | 6A | 1 | | |
| t | vvv | | | 5 | 001 | 6 | 011 | 6 | 101 | 7 | 111 | | | | | 2 | 010 | | |

## RTI

Description of RTI

Return to main program after an interrupt has been serviced. The status register and program counter are restored from the stack, and the stack pointer is adjusted.

Symbolic operation of RTI

↓ P
S+1 → S
↓ PC
S+2 → S

| Addressing mode | Opcode | n | t |
|---|---|---|---|
| Implied. | 40 | 1 | 6 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

B.D and I are also affected.

## RTS

| Description of RTS | Symbolic operation of RTS | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| Return to calling program from a subroutine. The return is to the instruction following the call (ie. following JSR). The program counter is restored from the stack and incremented by 1. The stack pointer is adjusted. | ↓ PC<br>S + 2 → S<br>PC + 1 → PC | Implied | 60 | 1 | 6 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

## SBC

| Description of SBC M | Symbolic operation of SBC M | Fixed bit pattern |
|---|---|---|
| Subtract the contents of M together with any borrow from the accumulator. The result is left in the accumulator and any borrow in the carry flag. | $A - (M) - \bar{C} \rightarrow A$<br>$\overline{\text{Borrow}} \rightarrow C$ | 111vvv01 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ |

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | E9 | 2 | E5 | 2 | ED | 3 | F5 | 2 | FD | 3 | F9 | 3 | F1 | 2 | | | E1 | 2 |
| t | vvv | 2 | 010 | 3 | 001 | 4 | 011 | 4 | 101 | 4* | 111 | 4* | 110 | 5* | 100 | | | 6 | 000 |

## SEC

| Description of SEC | Symbolic operation of SEC | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The carry flag is set to one | 1 → C | Implied | 38 | 1 | 2 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | 1 | – |

## SED

| Description of SED | Symbolic operation of SED | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The decimal flag is set to one. All arithmetic is now in BCD (ie. ADC and SBC operate according to BCD) | 1 → D | Implied | F8 | 1 | 2 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

D is set to 1.

## SEI

| Description of SEI | Symbolic operation of SEI | Addressing mode | Opcode | n | t |
|---|---|---|---|---|---|
| The interrupt mask bit is set to 1, so disabling interrupts. | 1 → I | Implied | 78 | 1 | 2 |

Flags affected

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

I is set to 1.

## STA

| Description of STA M | Symbolic operation of STA M | Flags affected | Fixed bit pattern |
|---|---|---|---|
| The contents of the accumulator is put into the location M. | A → M | | 100VVV01 |

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 85 | 2 | 8D | 3 | 95 | 2 | 9D | 3 | 99 | 3 | 91 | 2 | | | 81 | 2 |
| t | vvv | | | 3 | 001 | 4 | 011 | 4 | 101 | 5 | 111 | 5 | 110 | 6 | 100 | | | 6 | 000 |

## STX

| Description of STX M | Symbolic operation of STX M | Flags affected | Fixed bit pattern |
|---|---|---|---|
| The contents of X is put into the location M. | X → M | | 100VV110 |

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

| | | Immediate | | Zero Page | | Absolute | | Zero page Y | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 86 | 2 | 8E | 3 | 96 | 2 | | | | | | | | | | |
| t | vv | | | 3 | 00 | 4 | 01 | 4 | 10 | | | | | | | | | | |

## STY

| Description of STY M | Symbolic operation of STY M | Flags affected | Fixed bit pattern |
|---|---|---|---|
| The contents of Y is put into the location M. | Y → M | | 100VV100 |

| N | Z | C | V |
|---|---|---|---|
| – | – | – | – |

| | | Immediate | | Zero Page | | Absolute | | Zero page X | | Absolute X | | Absolute Y | | (Indirect) Y | | Accumulator | | (Indirect X) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OP CODE | n | | | 84 | 2 | 8C | 3 | 94 | 2 | | | | | | | | | | |
| t | vv | | | 3 | 00 | 4 | 01 | 4 | 10 | | | | | | | | | | |

## TAX

| Description of TAX | Symbolic operation of TAX | Addressing mode | Opcode | n | t | Flags affected |
|---|---|---|---|---|---|---|
| The contents of the accumulator is copied into the X register. | A → X | Implied | AA | 1 | 2 | |

| N | Z | C | V |
|---|---|---|---|
| ✓ | ✓ | – | – |

## TAY

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| TAY | TAY | | | | | N | Z | C | V |
| The contents of the accumulator is copied into the Y register. | A → Y | Implied | A8 | 1 | 2 | ✓ | ✓ | — | — |

## TSX

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| TSX | TSX | | | | | N | Z | C | V |
| The contents of the stack pointer is copied into the X register | S → X | Implied | BA | 1 | 2 | ✓ | ✓ | — | — |

## TXA

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| TXA | TXA | | | | | N | Z | C | V |
| The contents of the x register is copied to the accumulator | X → A | Implied | 8A | 1 | 2 | ✓ | ✓ | — | — |

## TXS

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| TXS | TXS | | | | | N | Z | C | V |
| The contents of the x register is copied to the stack pointer | X → S | Implied | 9A | 1 | 2 | — | — | — | — |

## TYA

| Description of | Symbolic operation of | Addressing mode | Opcode | n | t | Flags affected | | | |
|---|---|---|---|---|---|---|---|---|---|
| TYA | TYA | | | | | N | Z | C | V |
| The contents of the Y register is copied to the accumulator. | Y → A | Implied | 98 | 1 | 2 | ✓ | ✓ | — | — |

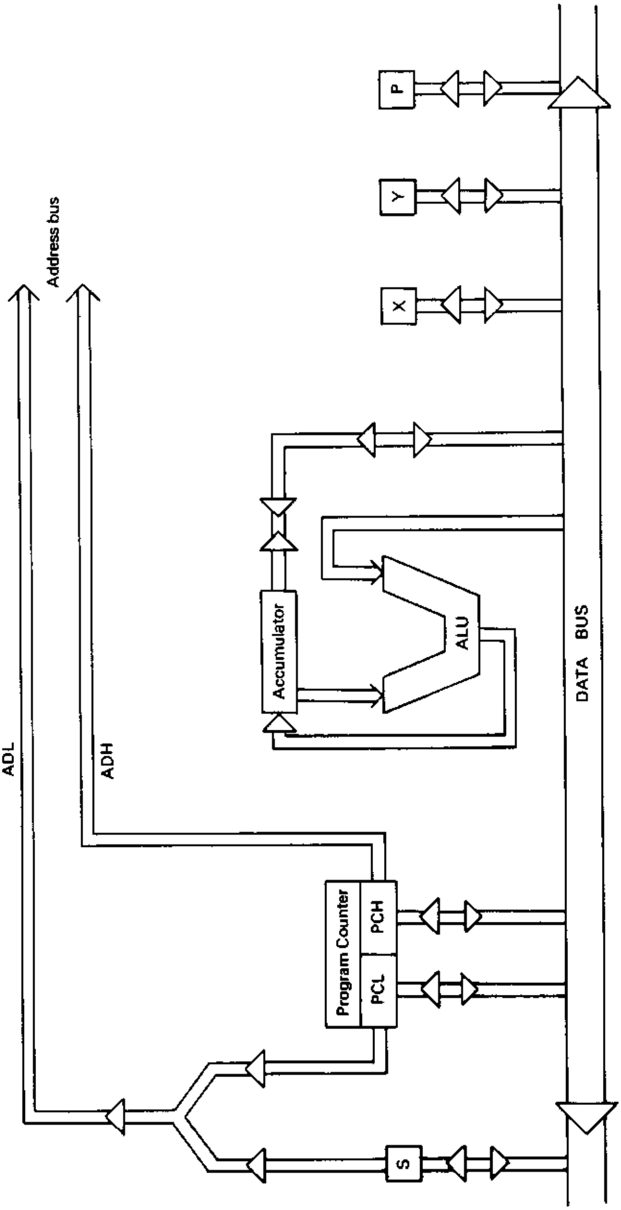
*TABLE A1.2: Instruction set in numerical order of opcodes*

In this table the following abbreviations are used:

| | |
|---|---|
| ZP | Zero page addressing mode |
| Abs | Absolute addressing mode |
| Imm | Immediate addressing mode |
| ZP,X<br>ZP,Y | Zero page indexed addressing mode |
| Abs,X<br>Abs,Y | Absolute indexed addressing mode |
| (Ind),Y | Indirect indexed addressing mode |
| (Ind,X) | Indexed indirect addressing mode |
| (Ind) | Indirect addressing mode |
| A | Accumulator addressing mode |
| LSN | Least significant nybble (e.g. A in &EA) |
| MSN | Most significant nybble (e.g. E in &EA) |
| – | Reserved for future expansion |

| →<br>↓ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | BRK | ORA<br>(Ind,X) | — | — | — | ORA<br>ZP | ASL<br>ZP | — | PHP | ORA<br>Imm | ASL<br>A | — | — | ORA<br>Abs | ASL<br>Abs | — |
| 1 | BPL | ORA<br>(Ind),Y | — | — | — | ORA<br>ZP,X | ASL<br>ZP,X | — | CLC | ORA<br>Abs,Y | — | — | — | ORA<br>Abs,X | ASL<br>Abs,X | — |
| 2 | JSR | AND<br>(Ind,X) | — | — | BIT<br>ZP | AND<br>ZP | ROL<br>ZP | — | PLP | AND<br>Imm | ROL<br>A | — | BIT<br>Abs | AND<br>Abs | ROL<br>Abs | — |
| 3 | BMI | AND<br>(Ind),Y | — | — | — | AND<br>ZP,X | ROL<br>ZP,X | — | SEC | AND<br>Abs,Y | — | — | — | AND<br>Abs,X | ROL<br>Abs,X | — |
| 4 | RTI | EOR<br>(Ind,X) | — | — | — | EOR<br>ZP | LSR<br>ZP | — | PHA | EOR<br>Imm | LSR<br>A | — | JMP<br>Abs | EOR<br>Abs | LSR<br>Abs | — |
| 5 | BVC | EOR<br>(Ind),Y | — | — | — | EOR<br>ZP,X | LSR<br>ZP,X | — | CLI | EOR<br>Abs,Y | — | — | — | EOR<br>Abs,X | LSR<br>Abs,X | — |
| 6 | RTS | ADC<br>(Ind,X) | — | — | — | ADC<br>ZP | ROR<br>ZP | — | PLA | ADC<br>Imm | ROR<br>A | — | JMP<br>(Ind) | ADC<br>Abs | ROR<br>Abs | — |
| 7 | BVS | ADC<br>(Ind),Y | — | — | — | ADC<br>ZP,X | ROR<br>ZP,X | — | SEI | ADC<br>Abs,Y | — | — | — | ADC<br>Abs,X | ROR<br>Abs,X | — |
| 8 | — | STA<br>(Ind,X) | — | — | STY<br>ZP | STA<br>ZP | STX<br>ZP | — | DEY | — | TXA | — | STY<br>Abs | STA<br>Abs | STX<br>Abs | — |
| 9 | BCC | STA<br>(Ind),Y | — | — | STY<br>ZP,X | STA<br>ZP,X | STX<br>ZP,Y | — | TYA | STA<br>Abs,Y | TXS | — | — | STA<br>Abs,X | — | — |
| A | LDY<br>Imm | LDA<br>(Ind,X) | LDX<br>Imm | — | LDY<br>ZP | LDA<br>ZP | LDX<br>ZP | — | TAY | LDA<br>Imm | TAX | — | LDY<br>Abs | LDA<br>Abs | LDX<br>Abs | — |
| B | BCS | LDA<br>(Ind),Y | — | — | LDY<br>ZP,X | LDA<br>ZP,X | LDX<br>ZP,Y | — | CLV | LDA<br>Abs,Y | TSX | — | LDY<br>Abs,X | LDA<br>Abs,X | LDX<br>Abs,Y | — |
| C | CPY<br>Imm | CMP<br>(Ind,X) | — | — | CPY<br>ZP | CMP<br>ZP | DEC<br>ZP | — | INY | CMP<br>Imm | DEX | — | CPY<br>Abs | CMP<br>Abs | DEC<br>Abs | — |
| D | BNE | CMP<br>(Ind),Y | — | — | — | CMP<br>ZP,X | DEC<br>ZP,X | — | CLD | CMP<br>Abs,Y | — | — | — | CMP<br>Abs,X | DEC<br>Abs,X | — |
| E | CPX<br>Imm | SBC<br>(Ind,X) | — | — | CPX<br>ZP | SBC<br>ZP | INC<br>ZP | — | INX | SBC<br>Imm | NOP | — | CPX<br>Abs | SBC<br>Abs | INC<br>Abs | — |
| F | BEQ | SBC<br>(Ind),Y | — | — | — | SBC<br>ZP,X | INC<br>ZP,X | — | SED | SBC<br>Abs,Y | — | — | — | SBC<br>Abs,X | INC<br>Abs,X | — |

# *Appendix 2 Full block diagram of 6502 architecture*



Block diagram of the 6502 microprocessor (with internal address pathways from the program counter omitted).

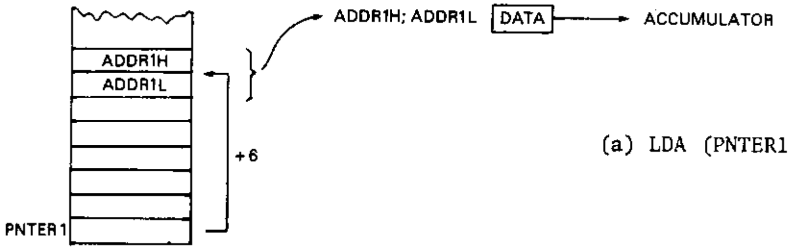
# Appendix 3 Indexed indirect addressing

There is one more 6502 addressing mode which we have not covered in the book: *indexed indirect addressing*. This omission is quite deliberate, for you are not likely to want to use this mode of addressing in your work on the BBC Computer. The designers of the 6502 included indexed indirect for a very specific purpose: *multiple peripheral programming.*

In this appendix we shall first describe briefly the operation of the indexed indirect mode, and then describe the sort of application for which it is suitable.

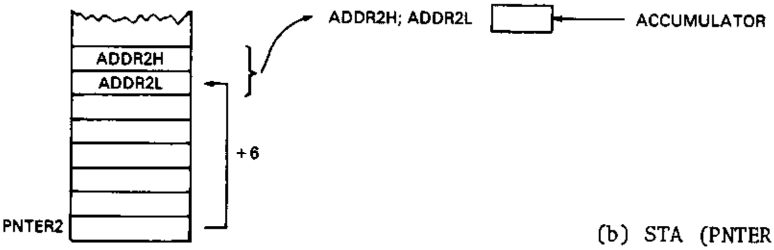
Consider a list of pointers stored consecutively in memory. Then the contents of the location to which any pointer is referencing can be loaded into the accumulator by writing LDA (PNTER,X), where PNTER is the base address of the pointers and X is a suitable even number. Similarly STA (PNTER,X) stores a copy of the accumulator in the location pointed to by (PNTER,X). Only the X register can be used for this purpose, just as only the Y register can be used for indirect indexed addressing. Again, like indirect indexed addressing, indexed indirect requires the location PNTER *to be in zero page.*

The diagram below illustrates the pair of instructions LDA (PNTER1,X): STA (PNTER2,X), where X equals 6.



(a) LDA (PNTER1,X)

(a) LDA (PNTER1,X)



(b) STA (PNTER2,X)

(b) STA (PNTER2,X)

6, the contents of X, is added to the base address PNTER1 to give the location containing the low byte of the address, the contents of which is ADDR1L. The next location will always contain the high byte of the address, in (a) ADDR1H. Hence the contents of the address ADDR1H; ADDR1L are put into the accumulator. Similarly, in (b) the accumulator is put into ADDR2H; ADDR2L.

Now, the indexed indirect addressing mode cannot be usefully used to access strings in memory by referring to a list of pointers to those strings, such as we constructed in Section 7.6. The reason for this is that each character of the string could then only be accessed by using ordinary indexed addressing, and we cannot combine both modes in the same instruction. That is, we cannot write LDA (BASE,X),Y (it is unfortunate we cannot do this - it would be a most powerful combination). Because most lists of pointers in the sort of assembly programs we are likely to write will be used to access a base address from which we will index, indexed indirect is not of much use to us. The methods used in Section 7.6 are still the best ones to use in these cases.

However, it is worth understanding the sort of applications where indexed indirect is useful - the sort of application, indeed, for which the addressing mode was designed. Consider a whole series of peripheral devices, say teletypes, each of which will be serviced by one 6502 microprocessor. Each teletype is connected up to its own specific VIA port (see Appendix 7). Each teletype is sending a message to the microprocessor, which will be terminated by a carriage return. As each character of the message is ready to be processed, the VIA to which the teletype is connected will interrupt the 6502. At this stage, the microprocessor will enter an interrupt service routine which will interrogate in turn the status registers of the VIAs to see which teletype has sent a character. Since teletypes are very slow, the order of polling is of no consequence: any multiple interrupt would be dealt with quite transparently to the user of any of the teletypes.

Each teletype has a small section of memory reserved for it which acts as a buffer. We shall assume here that the buffer is never overfilled. Five sets of pointers are required for this system, the order for the set of tables being arbitrary, although the order for each table must be the same.

(a) TABLE1 The addresses of the input register for the incoming character from the teletype.

(b) TABLE2 The addresses of the buffers for the teletypes.

(c) TABLE3 The addresses of output to the teletypes for acknowledgement.

(d) STATUS Addresses of the status registers for the teletypes.

(e) COPYTABLE2 A copy of TABLE2.

(a)-(d) must be in zero page, (e) can be anywhere. The function of TABLE3 needs to be explained. When a key is pressed on a teletype, the action of the print-hammer is caused not directly by this key but by the computer reflecting the key. In this way, an instant verification is performed that the correct data has been received.

Here, now, is the simplified code for performing this polling sequence (we simplify by ignoring parity checking, among other things):

```
 1              LDX #2 * (NUMBER - 1)
 2        LOOP1 LDA (STATUS,X)
 3              BMI STOREDATA
 4              DEX
 5              DEY
 6              BPL LOOP1
 7              BMI OUT
 8    STOREDATA LDA (TABLE1,X)
 9              STA (TABLE3,X)
10              CMP #&0D
11              BEQ ENDSTRING
12              STA (TABLE2,X)
13              INC TABLE2,X
14              BNE OUT
15              INC TABLE2+1,X
16              BNE OUT
17    ENDSTRING LDA COPYTABLE2,X
18              STA TABLE2,X
19              LDA COPYTABLE2+1,X
20              STA TABLE2+1,X
21              JSR ANALYSE
22          OUT      .
                     .
                     .
                     .
                     .
```

In line 1, NUMBER is the number of teletypes connected. The beauty of this program is that it will work for up to 32 teletypes, with no change whatsoever required in the program (32 × 2 × 4 = 256, the limit of zero page).

In lines 2 to 7 we examine the status register of each VIA in turn, beginning with the one at the top of the STATUS list. If bit 7 is 1, this indicates that the teletype to which this VIA is attached has sent a character, and we go to line 8 to process it. Otherwise, we decrease X by 2, and look at the next status register down the list. If we happen to go through the entire list without finding the source of the interrupt (a 'phantom' interrupt) we go to some suitable exit code at line 22 onwards.

Assuming we find the appropriate teletype, we then load the contents of the input and store it in the buffer (lines 8 and 9). Notice how indexed indirect allows us to recover the appropriate set of pointers by using just one index value (this is why the tables must be arranged in the same order, of course). If we have reached the end of the message (lines 9 and 10) we go to perform some analysis in 17-21. Lines 17-20 reset the pointers in TABLE2, which have been altered in lines 13-15, and line 21 jumps to a subroutine which performs some analysis on the basis of the message (and which will output a line feed, when the analysis is complete). During this analysis, interrupts will be enabled so that further input can be received. One function of the ANALYSIS routine will be to deal with the case where 2 or more teletypes have messages to be analysed simultaneously, using some time-sharing principle which need not concern us here.

If the end of the message is not yet reached, the current character will be stored in the buffer (line 12) and then the address of the buffer will be incremented by one to point to the next free space. This is slower and more cumbrous than the indirect indexed method used in 7.6, but it is suitable in this case since teletypes are relatively slow anyway, and relatively few incrementations are required (at least compared to the sorting requirements of 7.6).

The size of this program is very small considering the complex task it performs, and this is due entirely to the use of the indexed indirect mode. The overall speed of processing is very favourable too, and this is why the designers of the 6502 included this addressing mode. Unfortunately, we are unlikely to be able to profit from it on the BBC computer. About the only time we are likely to use it, is in the case where we want a simple indirect mode and the Y register is not available. In this case, using indexed indirect with X equal to zero will suffice, since LDA (BASE),Y and LDA (BASE,X) give identical results when X and Y are both zero.

# Appendix 4 Floating point representation

In this book we have only considered the integer (or fixed point) representation of numbers. The discussion of the *floating point* representation has been outside our scope. However, for the sake of completeness, we will here discuss this representation, although we will not be considering how arithmetic may be performed upon such numbers.

The number four in base two is 100; if we divide by two we obtain 10, or more suggestively 10.0; divide by two again and we get 1.0, which is one, of course. Now it would seem reasonable to write the result of dividing by two again as 0.1, by two yet again at 0.01, and by two still again as 0.001; and so on. Hence 0.1 is $\frac{1}{2}$, 0.01 is $\frac{1}{4}$, 0.001 is $\frac{1}{8}$; and so on. This is *bicimal* representation, the direct counterpart to the base-ten decimal; and we refer to the point as the *bicimal point*.

Any decimal can be written in bicimal; and any bicimal in decimal. For example, 0.75 is 0.11 in bicimal; and 0.0101 in bicimal is 0.3125. Now, fractions which can be written as terminating decimals may give recurring bicimals. For example $\frac{1}{5}$ is $0.\dot{0}01\dot{1}$ in bicimal. However, any fraction which terminates in bicimal will terminate in decimal, because all such fractions will have a denominator of a power of two, all of which terminate in decimal (just keep halving 0.5 until you get there). It follows that there may be a loss of precision in translating from decimal to bicimal if we cannot use the recurrence notation (the dots over the relevant repeating digits). Moreover, bicimal takes up many more places than decimal, so we may have to round to get our decimal into a fixed number of bicimal places. Hence, we see that a possible error can be introduced in translating from decimal to bicimal (and vice versa if the number of significant figures allocated to decimal output is fixed). This must be borne in mind when dealing with floating point numbers (in assembler *or* in BASIC), for in certain circumstances these rounding errors can compound considerably, resulting in significant errors.

When storing bicimal numbers in a computer it is convenient to first write them in a *normalised form*. So, we write 11011.01011 as $0.1101101011 \times 2^5$, and 0.00010101011 as $0.10101011 \times 2^{-3}$, for example. The convention is to move the bicimal point until the most significant digit is the one just after the point: that is, move it right or left until all digits to the left of the point are zero and the first digit to the right of the point is one. The power of two attached to this adjusted number reflects the number of moves that the bicimal point has had to make. Applying this power to the adjusted number will set the bicimal point back to its correct place (5 places rightwards in the first case, i.e. 0.1101101011 → 11011.01011 as required, and 3 places leftwards in the second). Thus the point is allowed to *float* across so that a normalised form is achieved, and so we call the representation the *floating point representation*. Whole numbers can also be represented in this way, of course. For example, 110001011 is 0.110001011 × $2^9$.

The BBC micro, and most others, use 5 bytes to represent such numbers. The least significant byte represents the power, or as it is usually called, the *exponent*. The next four bytes represent the number, or as it is usually called, the *mantissa*. The exponent is in two's complement form with one difference: the sign bit is reversed. Hence an exponent of &90 represents &10 or 16, whilst &70 represents -&10 or -16. The reason for this is connected with the representation for zero. Clearly zero gives a zero mantissa (which cannot be normalised since there is no one). It is logical to have the minimum exponent associated with this, which is the maximum negative exponent. This is reasonable since a maximum negative exponent is associated with the smallest number which can be represented for any given mantissa. Without the change in the sign bit this would give &80 00 00 00 00; with the change it gives &00 00 00 00 00, much more sensible.

Apart from zero, *all* mantissas will have their most significant bits as one. We can therefore *assume* that the most significant bit is one, and use the actual bit in this position to reflect the sign of the number: 0 is positive, 1 is negative.

Figure A4.1 shows the format for a floating point number: notice that the byte on the extreme left (carrying the exponent) is the *lowest* in memory of the five. Moreover, in the next four, the most significant byte is *lowest* in memory, and the least significant, *highest* in memory. This is in distinction to integer (fixed point) numbers, where the most significant byte is the highest in memory. The convention with integers is chosen to fit in with the 6502 convention; with floating point, there are standard routines for arithmetic and there is no gain in using the specific convention of the 6502 microprocessor.

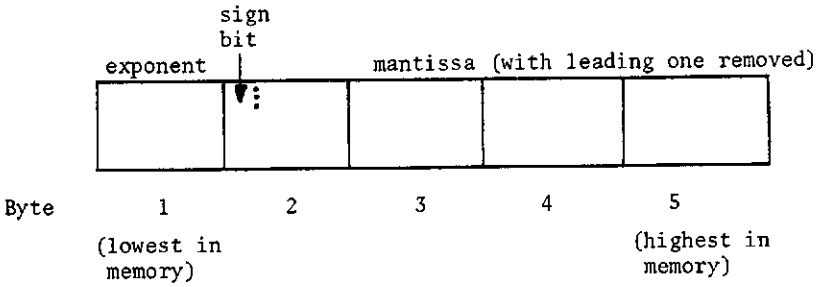


*Figure A4.1: Floating point storage in the BBC Computer*

The largest numerical values are, on the positive side FF 7F FF FF FF and on the negative FF FF FF FF FF (i.e. ±1.70141183 × $10^{38}$ to 9sf, the limit of precision in the BBC computer). The smallest numerical values (apart from zero) are 00 00 00 00 01 and 00 80 00 00 01 (i.e. ±1.46936794 × $10^{-39}$ to 9sf).

In order to acquaint yourself with this representation, load the monitor (program 10.6) and use CALLS%,X, with X set at various values. For example:

```
X = 2500: CALLS%,X
```

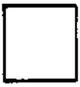
Now type M 0600 and the address pointed to should be &25E4. M 25E4 gives 8C 1C 40 00 00, which is &0.9C4 × $2^{12}$ (i.e. 0.1001 1100 0100 × $2^{12}$). Now moving the hexadecimal point across each hex digit is equivalent to multiplying by $2^4$, so we obtain &9C4, which is indeed 2500.

Again, X = -2500: CALLS%,X, gives 8C 9C 40 00 00 i.e. -&0.9C4 × $2^{12}$ or -&9C4, as required. Remember, the sign bit of the second byte is the sign of the whole number.

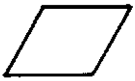
Finally, X = 0.3175: CALLS%,X gives 7F 20 00 00 00, which is &0.A0 × $2^{-1}$ i.e. 0.0101, as required.

Try more yourself - you will soon become very familiar with this form of storage.

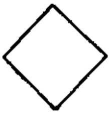
# Appendix 5 Flowchart symbols and conventions used in this book



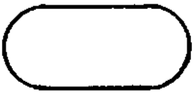
Assignment of values



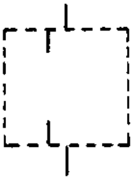
Input/output



Decision



Start or stop



Structured flowchart symbol:

(i) FOR..... NEXT

(ii) REPEAT..... UNTIL

(iii) REPEAT WHILE

# *Appendix 6 Linking programs on the BBC Computer*

## Ⓐ THE USE OF PAGE

In one sense, PAGE seems a pretty simple statement to use. Just type PAGE = &2000, say, load your program from tape, or type NEW and input a new program from the keyboard, and there you are. If you have another program at &E00, which isn't too big, it will still be there when you want it. In this way, you can store as many programs in memory as your computer can hold.

But there is more to PAGE than this, as these simple experiments will show:

In &E00 put

```
10      A$ = "It works!"
20      PAGE = &2000
30      GOTO 10
40      P.~TOP
```

Type PAGE = &2000, NEW, and input

```
10      P. A$
20      B$ = "It still works!"
30      PAGE = &3000
40      GOTO 10
```

Finally type PAGE = &3000, NEW and input

```
10      P. B$
20      PAGE = &E00
30      GOTO 40
```

Now follow these instructions, in each case noting what appears on the screen:

(a) Type PAGE = &E00, RUN

(b) Type OLD, RUN

(c) Type PAGE = &2000, OLD, PAGE = &E00, RUN

(d) Now press the ESCAPE key and type P.~TOP

(e) Type PAGE = &2000, OLD, PAGE = &E00, 50 REM, P.~TOP, LIST, PAGE = &2000, LIST

(f) Finally type ?&2000 = &D: ?&2001 = 0: ?&2002 = 1: ?&2003 = 5: ?&2004 = &41, and LIST.

We can draw the following facts from these experiments:

(i) Changing the page does *not* change TOP, the first free location where variables can be stored (cf (a) and (c)).

(ii) TOP is reset to the top of the program in the current page if we type OLD, or LIST, or ESCAPE (or with any program error).

(iii) Programs can share the same variables in a common TOP (cf (a), (b) and (c)).

(iv) When adding new lines to a program, it is PAGE which dictates where the new line will go, but if TOP does not match, a program in another page will have its pointers fouled up. Cf (e) and (f), where we cause the program in PAGE &2000 to move up 5, so it begins at &2005. Of course, if you want this to happen, the technique is useful, but...

Hence, we see that great care must be taken with PAGE. The most usual use of PAGE is to store lots of programs in memory and to pass manually from one to the other. The safest way to do this is *always* to leave enough space at the top of each program for all the dynamic variables and machine code. Then, always reset TOP by using OLD.

For example, suppose one program is in &E00, a second in &2800 and a third in &4000. Program the soft keys as follows: $f_0$ — PA.= &E00||M OLD||M RUN||M; $f_1$ — PA. = &2800||M OLD||M RUN||M, etc. Now you can pass effortlessly and safely between programs by pressing the appropriate programmable key.

A second use of PAGE is to chain between programs in memory, sharing variables. This is useful, but needs great care. You might be tempted to feel that you can use PAGE to add procedures and subroutines to a program. However, this is fraught with difficulty; in any case, there is a much better way as we shall now see.

## Ⓑ THE USE OF *LOAD

There is a very simple way of appending one program onto another. Here is the method:

(a) Make sure that the program to be appended is stored on tape with all its line numbers greater than the program in the computer to which it is to be appended. With procedures, and subroutines (BASIC or assembler), this is most conveniently done by creating a library with all the line numbers beginning at 30 000. Do not allow line numbers greater than this in your normal programs. (Actually, you *can* append programs with lower line numbers than the program in the computer, but any GOTO's, IF THEN's or GOSUB's will not be renumbered correctly. Indeed, the RENUMBER command will even cope with line numbers which are the same! However, it is safest to follow the advice given above.)

(b) Type P.∿TOP-2, and note the value given.

(c) Type *LOAD " "  XXXX, where XXXX is the number in (b), and load up the subroutine from tape.

(d) Renumber if necessary.

(e) Type LIST or OLD to reset TOP. It is essential to reset TOP when using *LOAD with BASIC programs. However, if you renumbered in (d), TOP will already have been reset, and so this precaution will be unnecessary in that case.

Reference to Section 10.2 on the structure of a BASIC program should help you to understand why this works.

# *Appendix 7 The user port*

The user port on a model B is part of a 6552 Versatile Interface Adapter (VIA). This is a fairly complex input/output chip, which provides two ports, handshaking, interrupts, two timers and a serial register. It is quite possible to write a book on the applications of this chip, and there is not space here to do this. However, information on the VIA in accessible and digestible form is not easy to come by, and in this appendix, a brief but full account is given of its workings. This should allow you to do most of what you want with the user port.

## A7.1 OVERVIEW OF THE 6522 VIA

(This section should be read quickly on first reading, and returned to later.)

The VIA which is used to create the centronics printer port and the user port is memory mapped into the locations &FE60 to &FE6F. Table A7.1 shows the purpose of each location. Since *port A* is used exclusively for the printer, we shall not consider it or its associated control lines in any detail (though if you require up to 10mA of buffered 'sink' current, port A could be used without modification to the circuits).

*Port B* at &FE60 has each of its bits connected to a corresponding pin on the user port. There are also two *control lines,* CB1 and CB2, which are connected to the other two pins on the user port: your manual contains the relevant circuit diagram. Each bit of the port can either be used for output or for input: the *data direction register* at &FE62 controls this.

The 6522 contains two *timers, timer1* being more complex than *timer2,* and both are available for use. The VIA supports interrupts on both these timers, and also on each of the control lines (CB1 and CB2 for the user port). There is also a *serial register* which can output and accept bits one at a time and shift them accordingly, and this can also generate interrupts. In the BBC computer all user port interrupts vector through an address contained in &0204 and &0205 (i.e. JMP (&0204) is performed). This means that the user can write his own interrupt service routines in connection with port B, the serial register and the timers. The *interrupt enable register* (&FE6E) controls which interrupts are allowed: the *interrupt flag register* (&FE6D) displays which items are calling for an interrupt (regardless of whether they are enabled), and bit 7 of this register is one if an interrupt is asked for and enabled. Tables A7.2 and A7.3 show these registers.

There are two control registers: the *peripheral control register* is concerned with the operation of the four control lines (CA1, CA2, CB1, CB2); the top nybble controls CB1 and CB2, the bottom CA1 and CA2 (which are reserved for the printer port). The *auxiliary control register* determines how the input ports, serial register and the timers behave. Tables A7.4 and A7.5 give the details.

## A7.2 CONFIGURING THE 6522 FOR INPUT/OUTPUT

Each bit of each port of a VIA can be programmed to act as an input source or an output source. The data direction registers at &FE62 and &FE63 are used to specify this. A *one* in the relevant position in the data direction register specifies *output* for the corresponding bit in the port. A *zero* in the data direction register specifies *input* for the corresponding bit in the port. Using zero for input is a safeguard, for momentary power failures, faults, resets etc. usually zeroise memory locations, and random output is far more dangerous than random input.

Location &FE63 contains &FF, since all the bits on port A are to be outputs to the printer and this location should not usually be touched by the programmer. Location &FE62 controls the user port and is at the disposal of the programmer. Here are some examples:

(a) All bits inputs:

```
LDA #0
STA &FE62
```

(b) Bottom nybble outputs, top nybble inputs:

```
LDA #&0F
STA &FE62
```

(c) Odd bits outputs, even bits inputs:

```
LDA #&AA
STA &FE62
```

(d) Bit 7 output, rest inputs:

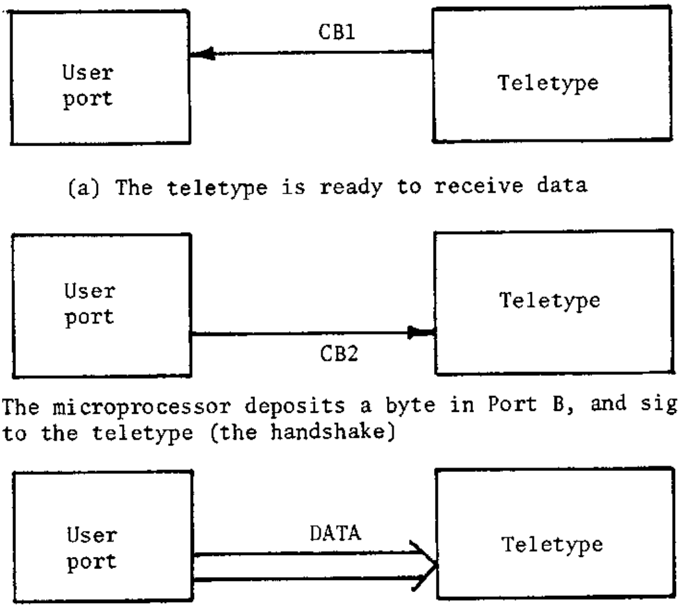
```
LDA #&80
STA &FE62
```

You can read the contents of port B even if one or more of its bits are designated as outputs i.e. LDA &FE60 will always give a valid reflection of the contents of &FE60. This is not true of port A, however, where the bits can be validly read only if they are designated inputs - fortunately, port B and not port A is the user port (port B is also a more powerful driver, and with suitable circuits, can drive solenoids etc.).

## A7.3 HANDSHAKING

Suppose we wish to send data to a teletype. The teletype has a parallel buffer which can store 8 bits, we shall assume. So we configure the userports to be all output (i.e. LDA #&FF: STA &FE62). We deposit the byte we wish to send into port B (STA &FE60, assuming the byte is in the accumulator), wait until the teletype has processed the byte, and then send the next one. But how do we know when the teletype is ready? And how does it know when we are ready to send the next byte? The answer lies in the concept of *handshaking*.

Port B has two control lines, CB1 and CB2. CB1 is *always an input,* and so will be used to transmit the signal from the teletype; CB2 can be an input or an output, and in this case we will use it as an output (how we specify this will become clear in a moment).



(a) The teletype is ready to receive data

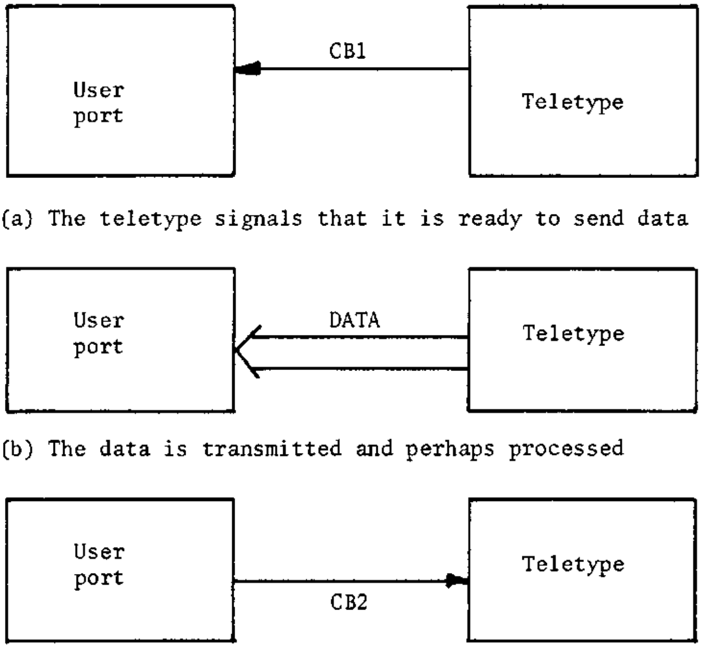
(b) The microprocessor deposits a byte in Port B, and signals to the teletype (the handshake)

(c) The data is transmitted and processed (e.g. a character is printed)

*Figure A7.1: An output handshake*

Refer to Figure A7.1. The idea is that, when the teletype is ready to receive a byte, it sends a signal on CB1 to the user port (A7.1). The microprocessor now deposits the byte in port B and sends a signal on CB2 to the teletype indicating that output is now valid (A7.1b). The teletype reads and processes the byte (A7.1c), sends a signal on CB1 asking for the next byte (A7.1a), and so the process continues.

This method of establishing connections between the user port and the teletype (or any other peripheral) is called handshaking: the teletype extends its 'hand' (signal on CB1), and the user port extends its 'hand' in recognition (signal on CB2). Now the information can pass between them. Sometimes CB1 and CB2 are referred to as *strobes* in this context: a strobe is simply an input and/or output line which indicates the availability of data to be transferred or the occurrence of a successful transfer. A strobe is usually a short pulse, one or two cycles long.



(a) The teletype signals that it is ready to send data

(b) The data is transmitted and perhaps processed

(c) The microprocessor signals that data has been successfully received (the handshake)

*Figure A7.2: An input handshake*

Figure A7.2 shows the same idea when port B is used for input (perhaps again from a teletype). When data is ready to be sent, the teletype sends a signal on CB1 indicating that data is ready (A7.2a). The user port responds by reading in the data (A7.2b), and sending a signal on CB2 indicating that the data has been successfully read (A7.2c) - this is the handshake. Again the process continues, with the teletype signalling the next byte is ready for transmission (A7.2a).

Notice that for output the handshake takes place *before* the data is sent, but that for input the handshake takes place *after* the data has been read. This is because the handshake is always finalised by the microprocessor.

Now the user port needs to provide the following facilities for these handshaking activities (we focus here only on port B - port A has almost identical features, but it is reserved for the printer in the BBC Computer):

(a) To designate CB2 as an input or output line.

(b) To fix the *levels* of input of CB1 and CB2 if relevant (i.e. is a signal to be interpreted as high to low voltage - a falling edge, *negative transition* - or vice versa - a rising edge, *positive transition*?)

These and other functions are the purpose of the top nybble of the peripheral control register (&FE6C): Table A7.4 shows the 8 possible configurations of bits 5 to 7, which control CB2, and the two configurations of bit 4 which control CB1.

You will notice in this table reference to the interrupt flag register. Bit 3 of this register will be set to 1 if there is an active transition (as defined by the peripheral control register) on CB2 and bit 4 if there is an active transition of CB1. Table A7.3 shows the entire register, and we will consider other flags later.

The difference in Table A7.4 between the handshaking and independent input modes of CB2 lies in this: in the handshaking mode, reading or writing to port B will clear bit 3 of the interrupt flag register automatically (to make way for the next handshaking operation), whereas in the independent mode one can read or write to port B without the interrupt flag being cleared (this is useful if CB2 is being used for a purpose unconnected with what occurs on port B).

However, in our applications here we are interested in the output modes of CB2. The handshake output mode sets CB2 low when data is written into port B by the microprocessor, and sets it high again on an active transition of CB1. The pulse output mode sets CB2 low for one clock cycle following a write to port B (a brief strobe). The last two constant modes are useful if we wish to provide output signals directly under software control, independently of what occurs at port B.

Now let us consider our output application again. We begin with LDA #&FF: STA &FE62, to create an output port at B. We will assume that active transition of CB1 is negative. Thus we write:

```
LDA #&80
ORA &FE6C
STA &FE6C
```

to configure CB1 and CB2 as required (ORA then STA, so as to preserve the information for port A). The handshaking sequence is now:

```
1   BEGIN LDA #&10
2   WAIT  BIT &FE6D
3         BEQ WAIT
4         LDA OUTPUT
5         STA &FE60
```

Repeating this, with suitable changes in line 4, is all that is needed to output as much data as required. Lines 1-3 wait until CB1 goes active, signalled by bit 4 of the interrupt flag register being set. At this stage, CB2 is automatically high (a feature of handshake output mode). Line 5 sets CB2 low and also clears the CB1 interrupt flag, all automatically. We can now return to line 1 to wait for the next CB1 signal. The signal on CB2, negative transition, has automatically occurred at line 5, and no doubt the teletype will respond in due course.

Consider now the input function. We set port B to input, and configure CB1 and CB2 as before. Lines 1 to 3 of the handshake are as before. We then write:

```
4   LDA &FE60
5   STA &FE60
6   STA OUTPUT
```

We have to write the data back to port B in line 5 in order to activate the CB2 line: in handshake output mode CB2 is only activated on a write (port A does not have this limitation, however, CA2 being activated by a read or write).

One final point in this section. To guard against changes in input before the input port is read - this is especially important if the microprocessor is doing much more than just the wait sequence in lines 1 to 3 - it is important to hold the input stable. To achieve this, the VIA is provided with *latches* which can protect input from corruption by changes on the input lines. To set the latch on port B we write a one into bit one of the auxiliary control register (at &FE6B) before entering the handshake (this need only be done once). Hence we write LDA #&02: STA &FE6B. Other functions of the auxiliary control register will be considered shortly.

## A7.4 INTERRUPTS ON CB1 and CB2

If we consider the input application above, it is clear that it is rather wasteful of processor time to wait until the teletype sends its next byte. Few teletypes work faster than about 30 characters per second, so the microprocessor could be doing other things most of the time. One strategy is to inspect the CB1 flag every 1/50 second - we will see how this is done in the next section. Another is to obtain an interrupt on CB1.

This is easily done: the interrupt enable register at &FE6E is the relevant register - see Table A7.2. We need to set bit 4 to 1 to enable interrupts on CB1. Once this bit is set to 1, an interrupt will be generated as soon as bit 4 of the interrupt flag register is set. In this case bit 7 of the flag register will also be set - this is used by the interrupt servicing routine when it is polling the potential causes of interrupt (see Section 9.3). In the case of the BBC machine, if this bit is set, a test is made to see if bit 1 of the enable and flag registers are also set. If they are, the printer has caused the interrupt - if not, the user port service routine will be entered by a JMP (&0204).

Since bit 4 of the interrupt flag register may already be one, it is essential to clear it before enabling the interrupt. This can be done either by writing 1 to bit 4 of the flag register i.e. LDA #&10: STA &FE6D; or by reading port B i.e. LDA &FE60. Clearly the latter is slightly quicker, but not if taken together with the enable for we can neatly write LDA #&90: STA &FE6D: STA &FE6E, as we will see below (writing one into bit 7 of the flag register does nothing).

If your interrupt routine is going at location &0D00 onwards, you will write ?&0204 = 0: ?&0205 = &0D. Now all interrupts on CB1 (and on CB2, T1, T2 and SR) will go to a routine at &0D00. The routine will consist of lines 4-6, with 6 suitably amended and expanded if necessary, and will end with JMP &DF09 (this restores the registers and returns from interrupt).

The interrupt enable register can only be altered by writing ones into the relevant bit positions: *writing zeros has no effect at all*. To enable CB1 interrupts we must write a one into bit 4 with bit 7 *equal to one*: to disable CB1 interrupts we write a one into bit 4 with bit 7 *equal to zero*. Thus, to enable CB1: LDA #&90: STA &FE6E; and to disable: LDA #&10: STA &FE6E. These operations will only affect the CB1 enable - all other bits will be unaffected.

## A7.5 USING THE PROGRAMMED TIMERS

There are two timers in the 6522, and both are at the programmer's disposal. Timer2 is the easier, and we shall consider this first.

Timer2 has two uses: it can generate a single time interval or it can count pulses input to bit 6 of port B. Bit 5 of the auxiliary control register determines which (see Table A7.5). The timer's counter consists of two bytes: the low byte at &FE68 and the high at &FE69. Always load the low byte first: loading the high byte clears the interrupt flag and starts the timing operation.

Suppose we wish to create an interval of 10,000 (&2710) clock cycles and then generate an interrupt. Here is the coding:

```
0     LDA #&DF
1     AND &FE6B      Set bit 5 to zero
2     STA &FE6B
3     LDA #&A0       Clear T2 interrupt flag
4     STA &FE6D
5     STA &FE6E      Enable T2 interrupts
```

```
6     LDA #&10   }  Load low byte with &10
7     STA &FE68  }
8     LDA #&27   }  high byte with &27 and start
9     STA &FE69  }  the countdown
```

If desired, this configuration can be done from BASIC using the query (?) operator.

In the service routine at &D00, just before JMP &DF09 is encountered, the statement LDA &FE68 must appear: this clears the T2 interrupt flag.

Note that a clock cycle here is one half the 2 mHz machine cycle e.g. 10,000 cycles is 1/100th of a second. This is because the clocking is done by the phase 2 clock which times memory operations, and this runs at 1 mHz.

The pulse counting mode is used to access an external clock or to synchronise with a set of external events. Changing lines 0 and 1 above to LDA #&20: ORA &FE6B will count 10,000 pulses incoming on bit 6 of port B.

Timer1 has more interesting applications. Instead of generating just one time interval it can generate a whole series of intervals. Bit 6 of the auxiliary control register determines this (Table A7.5). When the high byte is loaded countdown will begin: if bit 6 of the auxiliary control register is 1, at the end of countdown the counter will be re-loaded with the original contents of the counters which are stored in latches (at &FE66 and &FE67), and countdown will begin again.

Consider the teletype input again. We can generate an interrupt every 1/50th of a second by loading timer1 with 20,000 (&4E20) in continuous mode:

```
0     LDA #&40   }
1     ORA &FE6B  }  Set bit 6 to one
2     STA &FE6B  }
3     LDA #&C0   }  Clear T1 interrupt flag
4     STA &FE6D  }
5     STA &FE6E     Enable T1 interrupts
6     LDA #&20   }  Load counter and latch with low byte
7     STA &FE64  }
8     LDA #&4E   }  Load counter and latch with high
9     STA &FE65  }  byte and start count
```

Again in the service routine at &D00 include LDA &FE64 to clear the T1 interrupt flag. And again, BASIC can be used for the configuration if desired.

It is possible to alter the contents of the latches while the countdown is proceeding without affecting it in its present run. On the next run, however, new contents will be loaded. This is particularly useful with the other feature of timer1, the generation of pulses out of bit 7 of port B (clearly this bit must be configured as output). In the continuous mode (bit 6 = 1), the level on bit 7 of port B will begin low, then at timeout will go high, then at timeout again low, etc. Hence it is possible to create complicated waveforms, if the contents of the latches are also changed.

If we want to generate considerably longer delays, we can build this into our interrupt service routine. Let lines 0-9 be as before, but suppose we require an operation to occur every 1 second. To do this, we will require to reserve one location for the service routine, say &8F. The idea is that we load it with 50 and decrement it every interrupt. When it reaches zero we perform the required operation.

As an example of this, let us write a small routine which swtiches logical colour 3 from flashing yellow-blue (actual colour 11) to flashing cyan-red (actual colour 14), the transition occurring automatically every second. We will configure as before (lines 0 to 9), stored but not yet called. We also require a location, say &8E, where we store the actual colour. Begin by loading it with &0B(11). At &D00 we put:

```
 1      INC &8F
 2      LDA &8F
 3      CMP #50
 4      BNE OUT
 5      LDA #19
 6      JSR &FFEE
 7      LDA #3
 8      JSR &FFEE
 9      LDA &8E
10      JSR &FFEE
11      EOR #&05
12      STA &8E
13      LDA #0
14      JSR &FFEE
15      JSR &FFEE
16      JSR &FFEE
17      STA &8F
18  OUT LDA &FE64
19      JMP &DF09
```

Notice the trick used in line 11 which switches bits 0 and 2 of &8E so that &0B → &0E and &0E → &0B. The beauty of this interrupt method is that the timer will decrement on its next cycle regardless of whether the last interrupt has been completely serviced. As long as the last interrupt is serviced before the next one occurs the timing will be almost exact.

Finally change &204 and &205 to 0 and &D, respectively, load &8F with zero, and call the configuring program (lines 0 to 9). The effects will be seen in all the colour modes: try mode 1 first.

## A7.6 THE SHIFT REGISTER

The 6522 has a shift register (at &FE6A) which will input or output bits one at a time under timed control. The timing can either be provided by an external clock, the internal machine clock or else timer2. Bits 2, 3 and 4 of the auxiliary control register determine which (Table A7.5).

If the teletype in our previous examples lacked a parallel buffer, it would be possible to input and output bits synchronously using timber2 and the shift register. To achieve handshaking, it will be necessary to choose the mode which disables the timer each time, and to provide a suitable set of interrupts. The software turns out to be quite tricky, and it is better to use a UART (such as the 6551 ACIA which does not need an external clock): this will sort out all the parity and framing errors, and provide the stop and start bits. This can generate its own interrupts, but the circuitry might be easier if connections are made to the user port, utilising the VIA's interrupts instead (by using the expansion bus). Of course, for a 300 baud teletype the computer's own RS423 port can be used, and handshaking here is very simple to implement.

There are really only two main uses for the serial registers as far as we are concerned. One is to provide a source of memory clock pulses, or to receive them from another computer, so as to achieve synchronisation between the computers. Use 010 or 110 at bits 2-4 of the auxiliary control register to achieve this. The other is to output a variety of square waves independently of microprocessor control. This allows frequencies from about 2 Hz

| *Address* | *Function* |
|---|---|
| FE60 | Port B |
| FE61 | Port A, with handshaking (printer) |
| FE62 | Data direction register for Port B |
| FE63 | Data direction register for Port A (printer) |
| FE64 | Timerl counter, low byte |
| FE65 | Timerl counter, high byte |
| FE66 | Timerl latch, low byte |
| FE67 | Timerl latch, high byte |
| FE68 | Timer2 counter, low byte |
| FE69 | Timer2 counter, high byte |
| FE6A | Serial shift register |
| FE6B | Auxiliary control register |
| FE6C | Peripheral control register |
| FE6D | Interrupt flag register |
| FE6E | Interrupt enable register |
| FE6F | Port A, no handshaking (printer) |

*Table A7.1 User port and printer VIA*

(loading the shift register with &0F and timer2 with &FFFF) to 500 kHz (loading the shift register with &55 and timer2 with 1). 0100 at bits 2-5 of the auxiliary control register is the configuration in this case.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| Set or clear control bit | T1 | T2 | CB1 | CB2 | SR | CA1 | CA2 |

1 = interrupt enabled; 0 = interrupt disabled (bits 0 to 6)

1 = writing a one sets that bit to 1 } (bit 7)
0 = writing a one sets that bit to 0 }

*Table A7.2   Interrupt enable register*

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| IRQ | T1 | T2 | CB1 | CB2 | SR | CA1 | CA2 |

Bit 7 is 1 if any of bits 0 to 6 are set to 1 in both this register and the interrupt enable register.

Bits 0-6 are set and cleared by the following operations:

| Bit | Set by | Cleared by * |
|---|---|---|
| 0 | Active transition on CA2 | Reading or writing &FE61 |
| 1 | Active transition on CA1 | Reading or writing &FE61 |
| 2 | Completion of 8 shifts (not in free-running mode) | Reading or writing &FE6A |
| 3 | Active transition of CB2 | Reading or writing &FE60 |
| 4 | Active transition of CB1 | Reading or writing &FE60 |
| 5 | Time-out of Timer2 | Reading &FE68 or writing &FE69 |
| 6 | Time-out of Timer1 | Reading &FE64 or writing &FE65 |

**NB.* Interrupt flags can also be cleared by writing 1 into the bit position.

*Table A7.3   Interrupt flag register*

|  |  |  |  |  |
|---|---|---|---|---|
| Bit 4: | 0 | Active transition on CB1 is high to low |  | Bit 4 of interrupt flag register is set on active transition |
|  | 1 | Active transition on CB1 is low to high |  |  |
| Bits 5-7: | 000 | CB2 handshake input mode | Active transition on CB2 is high to low | Bit 3 of interrupt flag register is set on active transition |
|  | 001 | CB2 independent " mode |  |  |
|  | 010 | CB2 handshake " mode | Active transition on CB2 is low to high |  |
|  | 011 | CB2 independent " mode |  |  |
|  | 100 | CB2 handshake output mode | Active transition on CB2 is high to low |  |
|  | 101 | CB2 pulse output mode |  |  |
|  | 110 | Constant low output on CB2 |  |  |
|  | 111 | Constant high output on CB2 |  |  |

*Table A7.4 Peripheral control register (bits 4-7 only; bits 0-3 are identical in function but are for port A)*

| | | | |
|---|---|---|---|
| Bit 0: | 0 | Disable input latch on Port A | |
| | 1 | Enable input latch on Port A | |
| Bit 1: | 0 | Disable input latch on Port B | |
| | 1 | Enable input latch on Port B | |
| Bits 2-4: | 000 | Disable shift register | All output on CB2 |
| | 001 | Shift in at Timer2 rate | |
| | 010 | Shift in at machine clock rate | |
| | 011 | Shift in at external clock rate on CB1 | |
| | 100 | Free-running output at Timer2 rate | |
| | 101 | Shift out at Timer2 rate | All input on CB2 |
| | 110 | Shift out at machine clock rate | |
| | 111 | Shift out at external clock rate on CB1 | |
| Bit 5: | 0 | Decrement Timer2 in single-interval mode using machine clock | |
| | 1 | Decrement Timer2 on external pulses via bit 6 of Port B | |
| Bit 6: | 0 | Single-interval mode on Timer1 | |
| | 1 | Free-running mode on Timer1 | |
| Bit 7: | 0 | Disable output via bit 7 of Port B - Timer1 only | |
| | 1 | Enable output via bit 7 of Port B - Timer1 only | |

*Table A7.5 Auxiliary control register*

# Appendix 8 Some important zero page locations

| *Location (Hex)* | *Purpose* |
|---|---|
| 00-01 | LOMEM |
| 02-03 | Top of dynamic variables in BASIC |
| 04-05 | Pointer to first free area in BASIC stack |
| 06-07 | HIMEM |
| 0A | (&0B) + &0A = current position in BASIC text, either in a program or in the input buffer (&700 onwards) |
| 0B-0C | Beginning of current piece of BASIC text |
| 12-13 | TOP |
| 18 | PAGE |
| 19-1A | As 0B, 0C, but used for expression evaluation and conversion |
| 1B | As 0A, but for 19, 1A |
| 2A-2D | Accumulator for integer work |
| 2E-35 | First floating point accumulator |
| 3B-42 | Second floating point accumulator |
| 42-47 | Reserved as a work area for calculation |
| 4B-4C | Location of current variable |
| 70-8F | Guaranteed free for user's assembly programs |
| D7 | Keyboard scan value of current key pressed |
| D8 | Shift/caps key lock:<br>10 sets shift lock<br>20 sets caps lock<br>30 releases locks<br>40 sets both locks |
| DB | Shift key pressed if equals FF; zero otherwise |
| FD-FE | Location of last error message output |
| FF | Negative of ESCAPE pressed. |

# *Appendix 9 Operating System Differences*

Almost all the information in this book is independent of the particular version of the operating system (OS) your computer uses; but there are a few major differences which need to be highlighted here between OS 0.1 and all later versions (i.e. OS 1.0 onwards). Use *FX0 to discover which OS you have.

## A9.1 INTERRUPTS

### (a) Vectored interrupts

OS 1.0 onwards supports a vectored interrupt on IRQ. After checking for BRK, the computer indirects to the interrupt service routine through &204 and &205. It is therefore possible to incorporate top priority interrupts into the system, testing for a particular device before jumping to the standard service routine. It is also possible to add one's own code to existing interrupts, but care must be taken to differentiate between all the possible sources of interrupt. In general, it is better to use the user port interrupt if possible. In all cases, however, *it is essential not to use CLI in your own service routines.* (Recall that when the 6502 responds to IRQ it puts the status register on the stack and then *sets the interrupt disable flag automatically*. The flag will be cleared on RTI when the old status register is recovered.)

### (b) Returning from interrupt

JMP &DF09 is specific to OS 0.1. In later versions, the equivalent is easily found by disassembly of the interrupt service routine; alternatively you may write your own return sequence, which will normally be PLA : TAY : PLA : TAX : PLA : RTI.

### (c) BRK

OS 1.0 onwards preserves the status register on the stack on a BRK, and saves the accumulator in &FC (rather than in &DE in OS 0.1).

## A9.2 MEMORY-MAPPED INPUT/OUTPUT ACCESS

It is not possible to access any first processor memory directly from a second processor down the Tube. In particular, this applies to the user port from &FE60 to &FE6F. OS 1.0 onwards provides an OSBYTE routine to accomplish access from the other side of the

Tube. Instead of LDA &FE60 one writes LDA #&96 : LDX #&60 : JSR &FFF4, and instead of LDY #&FO : STY &FE62 one writes LDA #&97 : LDX #&62 : LDY #&FO : JSR  &FFF4. In general one *reads* from &FEXX by putting &96 in the accumulator and XX in the X-register, and one *writes* to &FEXX by putting &97 in the accumulator, XX in the X-register and the value to be written in Y-register: in both cases the subroutine is at &FFF4.

If you do not think that the routine you write will be used on a second processor then there is no need to worry about this, but if there is any chance of your having the Tube at some future date, then to ensure compatibility, use this OSBYTE formation. (By the same token, if you intend to use the Tube then you will need to use OSWORD with zero in the accumulator to accomplish the equivalent of INPUT, instead of the interpreter's routine recommended in Chapter 6.)

## A9.3 TURNING OFF THE SCREEN WHEN USING THE PRINTER

In the high-resolution screen copy it was necessary to precede every output to the printer with LDA #1 : JSR OSWRCH to protect the screen display. OS 1.0 onwards provides an easier way to do this, using an OSBYTE call. One needs to execute the following code *once only* at the beginning of the program:

```
LDA #3 : LDX #2 : JSR &FFF4
```

At the end of the program one restores the screen with:

```
LDA #3 : LDX #0 : JSR &FFF4
```

Assembly Language Programming for the BBC Microcomputer

Two software cassettes are available to accompany this book. They cost £9.00 each, if bought separately, or £16.00 for both when ordered together.

TAPE 1

Contains all the listings in Chapters 2 to 9
*Plus* these two extra programs

GRAPHPLOT which draws up to two graphs in the highest resolution available on your computer. (If you have an Epson printer, you can also obtain a hard copy of the graphs)

*and*

DISASSEMBLER which will translate any section of machine code back into standard 6502 mnemonics. This program is written entirely in BASIC, so it can be loaded into any page.

Cassette 1 ISBN 0 333 34587 8 £9.00 (inc. VAT)

TAPE 2

Contains all the listings in Chapter 10 and in the Answer section
*Plus* these two extra programs

FINDCODE which will locate any section of code in a program and display all the lines containing that code

*and*

REPLACE which will locate any section of code and replace it by any other

Cassette 2 ISBN 0 333 35016 2 £9.00 (inc. VAT)

*These cassettes are available through all major bookshops ... but in case of difficulty order direct from*

Globe Book Services
Canada Road
Byfleet
Surrey KT14 7JL

£9.00 each
*or*
£16.00 for the two

Every BBC Microcomputer is equipped with a powerful assembler. Sooner or later you will want to learn how to use it: this book will teach you. No prior knowledge of assembly language programming is assumed, and the reader is taken from the basics to their complex implementation.

The book will appeal to three sorts of reader. First, all those owners of BBC Microcomputers (Model A or B) who want to extend their knowledge into machine code. To help them in their self-study, the author has provided many exercises together with *full* solutions. Second, the teacher or student of computer science who wants a text for a structured course; the book is based on the author's own experience and approach as a teacher. Third, those who already know how to program in BASIC and are considering buying a BBC Microcomputer. The author's demonstration of the powers of this machine's assembler and operating system will persuade many to take that decision.

The book contains some 73 listings, many of which are helpful utilities, such as a full machine code monitor, a suite of machine code sorting programs, a high resolution screen copy and a program compactor.

## Software Tapes

Two cassettes are available to accompany the book, each containing two additional programs not contained in the book. Further details of these may be found inside the back of the book.

**ISBN 0 333 34585 1**