Language of the DRAGON: 6809 ASSEMBLER

Mike James

Sigma Technical Press

LANGUAGE OF THE DRAGON

Mike James

Sigma Technical Press

ISBN 0 905104 35 6

Published by:
SIGMA TECHNICAL PRESS
5 Alton Road
Wilmslow
Cheshire
UK

Distributors:

Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester
West Sussex, England

Australia, New Zealand, South-East Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
GPO Box 859, Brisbane,
Queensland 40001, Australia

Typeset, Printed and Bound by Commercial Colour Press, London E7.

Preface

The6809 microprocessor that lies at the heart of the Dragon is one of the most logical and easy-to-program of today's microprocessors. This makes it particularly suitable for learning assembly language with the minimum of fuss and confusion.

The aim of this book is to introduce 6809 assembly language programming to anyone with a prior knowledge of Dragon BASIC. Starting off from first principles of machine code, the need to use assembly language is quickly established. However, rather than rely on a particular commercial assembler, the book presents its own BASIC assembler which is built up stage-by-stage. This approach has the advantage of making absolutely clear the details of what an assembler does by giving a practical demonstration of why each new facility is required and how each is added. Although the BASIC assembler can cope with any of the programs and routines in this book, its limitations (in terms of speed) are apparent when handling the longest of them. So, while the BASIC assembler provides a useful tool for coming to terms with assembler, and provides a good model if you wish to write your own, once you've mastered the assembly language needed to do so, it is not intended to be used extensively as it stands. So, if you do foresee making a good deal of use of an assembler, then at some stage you will probably want to consider buying one of the commercially available ones. The details of two such packages are given in Appendix II. If course, if you already have such a piece of software available, then it can be used in place of the BASIC assembler as you work through this book. It is still worth inspecting the BASIC listing, however, as it will reveal HOW an assembler works and some of the problems it has to overcome.

There are no short cuts to learning assembler and, unlike BASIC, where you can start to write simple and useful programs when you've mastered only

a few commands, there is a lot of ground to cover before you can do anything that seems at all impressive. It IS worth the struggle however as, at the end of the day, assembly language programming does bring all the promised benefits of increased power and speed. Above all, assembly language programming is a challenge which it is great fun to pursue.

Contents

Chapter One

Why Assembler?

The only way to get the maximum power from any computer is to program it in assembler. This is the only reason for using assembler as opposed to friendlier computer languages such as BASIC. Even powerful, up-to-date microcomputers such as the Dragon don't really run languages such as BASIC fast enough for anything other than small programs and applications where the user is prepared to wait. Until microcomputers become much faster there will always be the need to take advantage of the improved efficiency offered by assembler.

Of course, there are a number of secondary reasons why particular individuals might decide to learn assembler. For example, assembler brings you closer to the inner workings of your machine than any other language. Another reason that is not often acknowledged is that, if you are using computers for fun, then you might like to try something a little more challenging than BASIC! However, it is important to realise that, while there are endless personal reasons for choosing a particular language, the only practical advantage that assembler has is speed.

BASIC is easy

The transition from a high level language to assembler is bound to come as something of a shock. For a start, a high level language lets the programmer takes a good deal for granted and is therefore very much more compact than assembler. One of the first things you now have to realise is that one line of BASIC is often equivalent to several lines of assembler. There is no denying that assembler is more difficult to learn and use than BASIC. If this were not the case, then all personal computers

would come equipped with assembler instead of the standard BASIC. The fact that assembler IS more difficult should not put you off the task of learning it even if you are a relative newcomer to computing. BUT, if you don't already know BASIC then do not attempt to learn assembler until you do - assembler isn't a good first language! For the rest of this book it will be assumed that you can write BASIC programs on the Dragon. If you have this knowledge of programming, then learning assembler is well within your reach. To make full use of assembler it is an advantage to know something of binary numbers and one or two other topics but it is better to leave these until they become necessary.

As assembler is a language that is closely tied to the design of the computer, so a knowledge of the internal workings of the Dragon is useful but not necessary. The need for assembly language often arises when you are trying to make creative use of some part of the machine's hardware where speed is essential. For example, the Dragon's sound generator can be controlled directly from BASIC but not fast enough to make any sounds other than a low pitched buzz - from assembler the range of sounds that you can make is unlimited. Many of the examples later in this book use hardware features of the Dragon and if you would like a fuller understanding of the overall hardware design, then you might like to consult the companion volume to this book, "The Anatomy of the Dragon", which is also published by Sigma. Indeed it may be that the difficulty inherent in using BASIC to control the Dragon's hardware directly, encountered so often in "The Anatomy of the Dragon", is what that has pulled you toward assembler.

Which assembler?

Although the term 'assembler' has so far been used as if it referred to a single computer language, this is unfortunately far from the truth. In fact, each different microprocessor or CPU (Central Processing Unit) used at the heart of a complete machine has its own particular assembly language. Notice that two different machines may use the same assembly language if they also use the same microprocessor. For example, both the BBC Micro and the APPLE use the 6502 microprocessor so they both recognise 6502 assembly instructions. But, in practice, assembly language programs tend to make use of particular features of the machine and so the chance of an APPLE assembly language program working on the BBC Micro is very

small. For this reason it is better to think of assembly language programs being completely machine specific. In other words, you are writing Dragon assembly language for the Dragon and no other machine (with the obvious exception of the very similar Tandy Color computer).

The microprocessor used inside the Dragon is a very advanced device called the 6809. Although the 6809 is advanced you shouldn't fall into the trap of thinking that 'advanced' implies difficult. In fact the 6809 is a very suitable microprocessor from the point of view of learning a first assembly language because it is a very logical and well designed device. An assembly language, in common with almost any language, becomes difficult to learn if it is full of 'exceptions' and special rules. As far as is possible, 6809 assembly language is based on a number of simple general rules and ideas and, once you have mastered these, the whole language seems very easy.

Machine code and assembler

Life is just a little more complicated than the last section would suggest. A microprocessor doesn't work in terms of assembly language as written by programmers but in 'machine code'. There is a lot of confusion about the relationship between machine code and assembler that is worth clearing up at this early stage.

As will be clear by the end of this book, computers work entirely in terms of numbers. The wordy commands of BASIC are not directly recognised by a machine. Instead there is a great deal of software in the system devoted to converting them to a sequence of numeric commands. These numeric commands form the only language the the machine can obey directly and are usually referred to as 'machine code'. And so, to program a computer directly, we have to make lists of numbers, with each number corresponding to a single fundamental operation. This is fine for the computer but what about the programmer? Imagine if all of the command words in BASIC (e.g. GOTO, IF, FOR etc) had to be written in terms of code numbers. Apart from making programs unreadable, it would take quite some time to remember which code number corresponded to which command. The point is, that while computers can only deal with code numbers, humans find it much easier to work with meaningful words. This is where assembler comes into the picture.

The fact that a computer needs all of its instructions in the form of numbers is something that we can do nothing about. However, there is nothing stopping us from writing our programs in terms of easy to remember 'command names' and then translating them into the numeric codes used by the computer. For example, the 6809 machine code instruction that plays the same role as GOTO in BASIC (i.e. it transfers control from one position in a program to another) is 14 but it is much easier to read and write the three letter command name JMP, which is short for JuMP. Such easy to remember command names are usually called 'mnemonics' because they help you to remember the commands. (Mnemonic derives from the Greek word "mnemon", meaning "mindful".) To this end long names are avoided in preference for three-, or at most four-letter abbreviations. The collection of mnemonic commands is called the machine's 'assembly language' and the act of converting it to the numeric codes is called 'assembling the program'.

You should now be able to see that there is such a close connection between machine code and assembly language that there is a tendency to treat them as the same thing. But there are important distinctions:

1) Machine code is composed of nothing but numbers and is the language that the computer obeys directly.

2) Assembly language is an easy-to-use form of machine code with the numeric codes replaced by short memorable command words called 'mnemonics'.

3) Assembly language must be converted to machine code before a computer can obey it. This conversion is called 'assembling the program' and can be achieved by a special program called an 'assembler'.

An Assembler

Producing an assembly language program has a number of stages, some of which are the same as the stages in producing a BASIC program. Obviously the first stage is to write the program, it then has to be converted to machine code, loaded into the machine and run. As a result

of the run, errors or other misbehaviour have to be noted, their cause found and then the whole cycle repeated after corrections have been made.

You can see that an important component of this production cycle is the conversion of assembly language to machine code. For a small program this can be achieved by the programmer looking up the numeric equivalents of each instruction in an appropriate table - such as the one given in Appendix I. This method of conversion is known as 'hand assembly' and if you are only going to write the occasional small assembly language program it is good enough. However it soon becomes tedious if you are writing any number of assembly language programs and error prone if the program is at all long.

The solution is to make use of the power of the computer to automate the process of hand assembly. Instead of you looking up the numeric code in a table why not write a program that does just that. Such a program is called an 'assembler'. So an assembler takes assembly language as its input and outputs machine code thus -

assembly language--> assembler --> machine code

This sounds like a good idea but where do you get an assembler from? There are a number of standard assemblers on the market and there is no doubt that if you are going to use assembly language often then it is worth investing in one of them. However, so that you can try out the ideas explained in this book without having to resort to tiresome hand assembly or an expensive assembler a simple assembler will be given using nothing but BASIC. Writing an assembler in BASIC may sound like a difficult program but, as you will see as things progress, it is fairly straightforward. To make it even easier to understand, rather than quoting a listing of the complete assembler, it will be built up chapter by chapter. In each case only the commands and facilities discussed in the chapter will be added and so, you should not only be able to understand the assembly language of the Dragon, you should also be able to build up a picture of how an assembler works. If you are not interested in how an assembler works, then you can skip the explanations and just use the program (a complete listing of which is given in Appendix II.) However, there is nothing like understanding the principles behind software for demonstrating how simple it really is.

Trial and error

In the next chapter the practical side of assembly language is introduced and it is important that you do try out the examples. So far, most of the ideas that have been presented come into the category of theory but it is surprising how a little theory can make practice seem easier. The reverse is also true and so, after you have reached Chapter Three or, Four try returning to this chapter and look over its contents. You might be pleased to discover that things fit together to form a complete picture! This principle of reading on and then going back is a technique for reading computer books that you should always apply. If you find that you are not understanding something don't immediately stop and go back to earlier material, carry on reading to the end of the chapter or section and THEN go back. It is often the case that later information clarifies earlier misunderstandings but going back too soon simply reinforces them!

Chapter Two

Registers and Operations

The microprocessor, or CPU, is the powerhouse of any computer. It is the place where all of the calculations and operations on data are carried out. From the point of view of the assembly language programmer, the CPU is also the most important part of the computer because it determines the operations that are possible and so dictates the form of the assembly language. However, even the all-important CPU has to work along with the other parts of the computer. In particular, it works in a close partnership with the computer's memory. In this chapter the idea of the CPU taking both its data and its instructions from memory is explained, some simple assembly language instructions are introduced and the first part of the simple assembler are given.

Memory, addresses and data

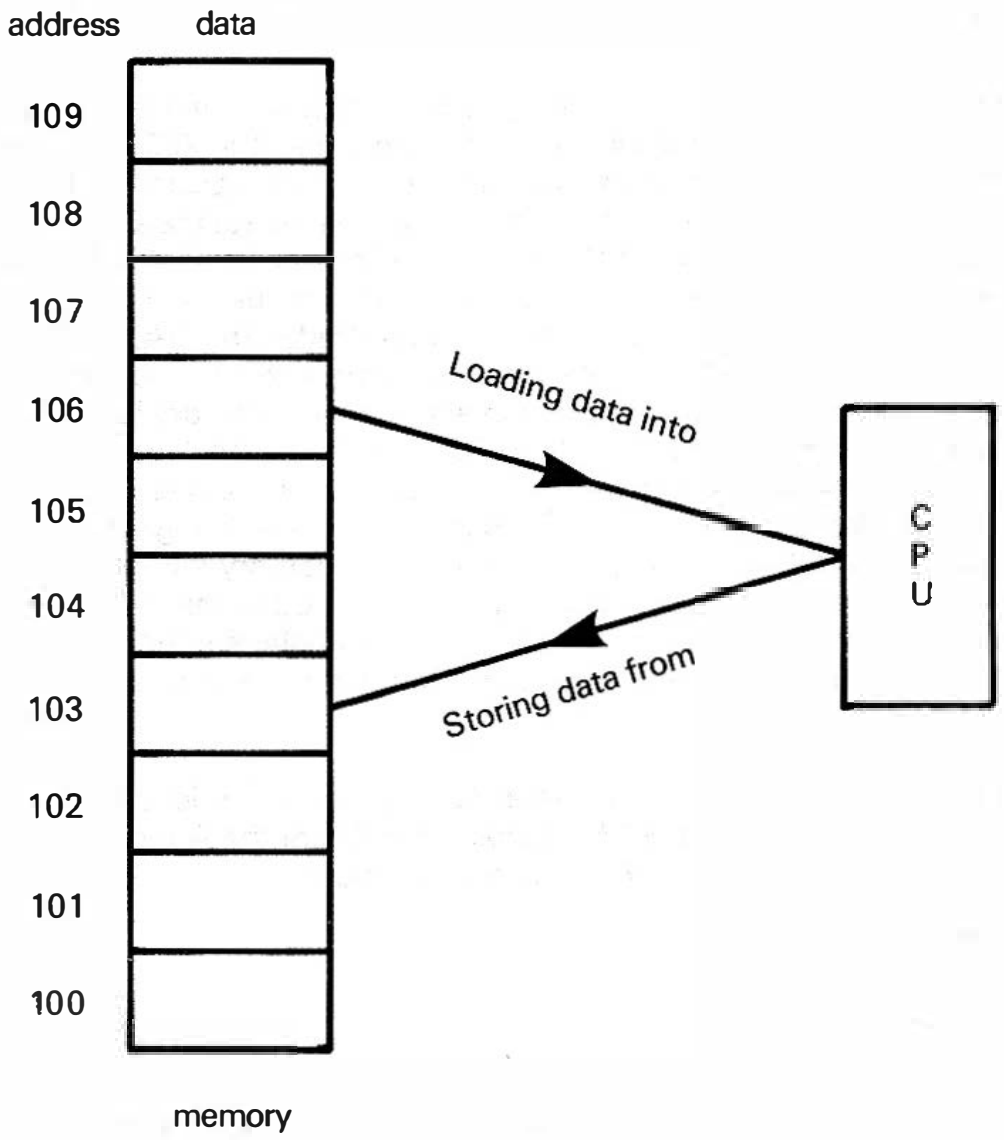
You probably already know that the best way to think of a computer's memory is as a large collection of boxes or pigeon holes each capable of storing a single number. These boxes or 'memory locations' are used by the CPU to store information. Obviously to be of any use it is important that each 'box' in the memory has a unique name so that the CPU can refer to the box that it wants to store something in or retrieve something from. This name is usually referred to as an 'address' and, as computers work only with numbers, it makes good sense to restrict ourselves to numeric addresses. Thus, each memory location has associated with it a

number, its address, and a number stored within it, its data. The Dragon's RAM occupies addresses from 0 to 32767 and each memory location can store a number in the range 0 to 255. The reason for the limits being 32767 and 255 are connected with the way that a computer actually stores numbers in binary and this will be discussed a little later in this chapter.

Now that we understand the way that memory works in terms of addresses and data, we need to ask what role the CPU plays. First it is important to realise that the memory can only contain data, either representing a computer program or the information used by the program. Any changes or operations on data are done within the CPU. Indeed, the memory cannot take the necessary steps to store and retrieve data without the aid of the CPU and in this sense it is best thought of as the CPU's slave. The relationship between the CPU and memory can be seen in fig 2.1 where a portion of memory is shown as a column of boxes and the CPU supplies the address of the box that it uses to either store or retrieve data. Fig 2.1 shows the data from memory being produced from the CPU and stored in memory or vice versa. This poses the question, what happens to the data once it is inside the CPU? In memory data is stored in a particular location. Where is it stored while it is inside the CPU? The answer to both of these questions lies in the study of the internal structure or, to use the accepted jargon, the 'architecture' of the CPU.

Registers - the CPU's own memory

The storage of data implies that some kind of memory is in use and so it is with the CPU. Within the CPU are a small number of very special memory locations called 'registers'. To be absolutely clear, registers have nothing to do with the computer's main memory and addressing methods. They are more like internal 'notepads' that the CPU uses to hold data while it is working with it. As there are only a small number of registers inside the CPU it is usual to give each one a name rather than a numerical address. There are nine registers inside the 6809 microprocessor that makes up the Dragon's CPU. Rather than examining them all in one go, it is less confusing to introduce them as the need arises. Perhaps the two most useful registers are the A and B accumulators. The reason why they are called accumulators is a little difficult to explain at this stage but, roughly speaking, an accumulator is a register where data cannot only be stored but can also be changed (i.e. the result is 'accumulated'). The ways



(only a small section shown)

Fig2.1 Memory and the CPU

in which data can be changed are fairly limited and it is this limitation that generally makes assembly language programming more difficult than say BASIC.

The 6809's A and B registers are just like ordinary memory locations in that they can each hold a single number in the range 0 to 255. This means that the contents of a single memory location can be transferred to either the A or the B register. The 6809 carries out the instructions LDA, short for LoaD the A register, and LDB, which is short for LoaD the B register, for just this purpose. So, when the 6809 comes to obeying the numeric code in a machine code program that is associated with LDA, it loads the A register, but from where? Obviously an instruction to load a register from memory must include information about which memory location is to be used. The complete form of the LDA is in fact LDA 'address' where 'address' is a number that is used as the address of the memory location whose contents are transferred (or loaded) into the A register. So, for example, LDA 421 would read the contents of memory location 421 and store them in the A register. It is important to notice that following the transfer the contents of memory location 421 is unaltered. Indeed the only thing that has changed is that the A register now has the SAME contents as location 421.

To change the contents of location 421, or any other location for that matter we have to use the STA (standing for STore the A register) or the STB (standing for STore the B register) instructions.

```
    STA 'address'
```

and

```
    STB 'address'
```

will store the contents of the A or B register in the memory location whose address is given by 'address'. For example STB 421 will store the contents of the B register into the memory location whose address is 421. Once again notice that following this instruction the B register is unaltered.

At this point it is worth clearing up some common misconceptions about memory. Whenever the Dragon is switched on every RAM location will contain some data - there is no such thing as 'empty' memory! The memory locations are initialised to contain random values, some of which are quickly changed to meaningful data by BASIC or whatever you are using. The value in a memory location is changed only by storing a new

value in it and, when that happens, the value that it formerly contained is lost forever - in other words storing a new value in memory (or a register for that matter) overwrites the old value. Moreover, retrieving (i.e. reading) the contents of a memory location (or again a register) doesn't alter its contents in any way - its contents are simply copied.

A short program

Now that we have four assembly language instructions LDA, LDB, STA and STB we can write a very simple program -

```
LDA 100
LDB200
STA200
STB 100
```

This program first loads the A register from memory location 100 and the B register from memory location 200 and then stores A in 200 and B in 100 so effectively swapping the contents of the two memory locations. Notice that even for this very simple operation of swapping the contents of two memory locations the data still had to be brought inside the CPU before anything could be done to it! This is such a simple program that hand assembling it is not too much trouble and is also quite instructive. The code that corresponds to LDA is 146, for LDB it is 214, for STA it is 147 and for STB it is 215. (If you have tried to use the table in Appendix I to look up the codes for LDA etc you will have noticed that the values it contains are all in hexadecimal. Don't let this worry you now. By the end of the next chapter you will be confident about using hex. You may also have been puzzled by the fact that there is more than one choice of code for each mnemonic. Again don't worry about this for the moment it will be fully explained in the next chapter!) So in machine code the program is

```
146100
214200
147200
215 100
```

and perhaps now you can see why programmers prefer to use assembly language mnemonics instead of machine code!

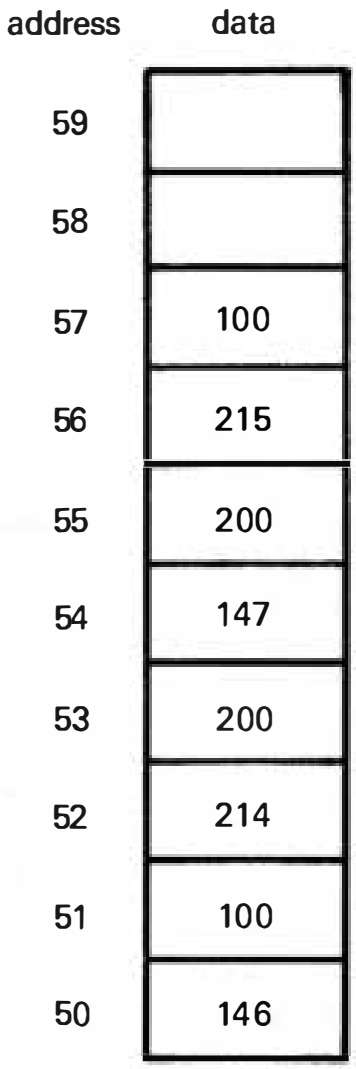


Fig2.2 The "swap" program stored in memory

The list of eight numbers given above is indeed a machine code program that the 6809 inside the Dragon will obey to swap the contents of memory locations 100 and 200. The next question that has be be asked is how does the 6809 ever get to 'see' the list of numbers that constitutes the program that we want it to obey? At the moment the list exists only on paper and the 6809 has no access to it! Obviously the program must be stored in memory as this is the only place that the CPU can obtain any sort of information from.

The fact that machine code program is stored in memory should not come as, any great surprise after all where else is a BASIC program stored? However, there are some important differences between the way a BASIC program and a machine code program are stored in memory. In particular the lines of a BASIC program are 'marked' by line numbers so that at any given moment the computer is obeying a particular line number but the commands of a machine code program are only 'marked' by the address of the memory location that they are stored in. Notice that a single instruction can occupy more than one memory location and so correspond to more than one address. For example the LDA 100 instruction takes two memory locations and is stored in 50 and 51. In obeying the program the 6809 would first carry out the instruction stored starting at 50 (i.e. the LDA 100) it would then move on to carry out the instruction starting at 52 and so on until the program was finished. (Notice that in this simple example the problem of stopping the computer at the end of the program has been totally ignored!)

You should be able to see that the address that an instruction is stored at can function in exactly the same way that line numbers do in BASIC. That is the 6809 can keep track of where it is in a program simply by remembering the address of the instruction that it is carrying out. Also, like a BASIC program, a machine code program is obeyed in order of increasing address, unless it is made to do otherwise by the machine code equivalent of a GOTO, a GOSUB or a RETURN. As you might guess, the address of the instruction that is being carried out is kept inside the CPU in yet another register - the PC or Program Counter - bringing the total number of registers that we know about to three. To make the 6809 obey the swap program the PC register would first be loaded with 50 (how, will be explained later) then the instruction starting at address 50 would be carried out and the PC register adjusted to point to the next instruction and so on until the program was complete.

The only thing that you have to be careful of is that the area of memory that is used to store the program isn't being used for something else. For example, it wouldn't be a good idea to store the swap program starting at memory location 100 (because this is one of the memory locations that is swapped and so we have to assume it contains data.) You should now be able to see that the Dragon's memory stores only numbers but these numbers can serve two different functions. They can be data that a program operates on or they can be instructions within machine code programs. Sometimes the CPU takes data from memory and sometimes it takes its next instruction.

Summary

So far the only machine code instructions that have been explained are LDA, LDB, STA and STB. However, a lot of new ideas have been introduced in this chapter, so it is worth gathering them together and attempting to summarise the main ideas:

1) Each memory location is identified by an address and can store a single number in the range 0 to 255.

2) Data is stored and manipulated inside the CPU using registers.

3) A machine code program is a list of numbers stored in memory; memory is, therefore, used to store both data and programs.

4) The address that an instruction is stored at is used in a way that is similar to a line number in BASIC. That is, it is used both to keep track of the current instruction and, as will be explained in the next chapter, as a way of referring to any instruction.

5) The address of the instruction being carried out is stored inside the CPU in the PC or Program Counter register. Instructions are normally carried out in order of increasing address.

6) The 6809 has two registers used to manipulate data, the A and B registers.

The BASIC Assembler

Although hand assembling the swap program was easy enough it is worth making a start on the BASIC assembler even at this early stage. So far, the only process that has to be carried out while assembling the swap program is to look up the commands such as LDA and replace them by their numeric codes. This is easy with a little BASIC program. The idea is to build a table of mnemonic codes and their corresponding numeric codes using a DATA statement and then, when a line of assembly language is typed in, the mnemonic code is compared to each entry in the table until a match is found. So far there are only four instructions to deal with but it is important to remember that the number will grow and so the BASIC assembler must be written so as to be easily expanded. Before trying to understand the program it is worth reminding yourself of the format of a line of assembly language -

> Each line starts with an optional space then there are a number of letters forming the mnemonic then at least one space followed by digits forming the address.

The program that follows will assemble the swap program given earlier if you type it in line by line. After the last line of the program type END and then the BASIC assembler will produce a listing of the machine code identical to that produced by hand assembly.

```
1 REM BASIC ASSEMBLER V2.1

10 DATA LDA,146,LDB,214,STA,147,STB,215,ZZZ,999

500 GOSUB 1000
510 GOSUB 2000
520 GOSUB 3000
530 GOSUB 4000
540 GOSUB 5000
550 GOSUB 6000
560 I = I + 1
570 IF I > TTHEN STOP
580 GOO 520

1000 DIM A$(50)
1010 I = 0
1020 P = 50
```

```
1030 RETURN

2000 INPUT L$
2010 IF L$ = "END" THEN T = I:I = 1:RETURN
2020 I = I + 1
2030 A$(I) = L$
2040 GOTO2000

3000 J = 1
3010 IF MID$(A$(I),J,1) = " " THEN J = J + 1:GOTO3010
3020 M$ = MID$(A$(I),J,1)
3030 J = J + 1
3040 IF MID$(A$(I),J,1)< > " " THEN
M$ = M$ + MID$(A$(I),J,1):J = J + 1:GOTO3040
3050 J = J + 1
3060 RETURN

4000 RESTORE
4010 READ C$,C
4020 IF C$ = "ZZZ" THEN ER = 1:GOTO9000
4030 IF C$ = M$ THEN RETURN
4040 GOTO4010

5000 A = VAL(MID$(A$(I),J))
5010 RETURN

6000 PRINT P;TAB(5);C;TAB(10);A
6010 P = P + 2
6020 RETURN

9000 PRINT "ERROR-";ER;"****"
9010 RETURN
```

After you have typed the BASIC assembler in it is important that you save it on tape because in subsequent chapters it will be improved on and added to. The program is written using BASIC subroutines for each job that the assembler must carry out. This makes later modification a matter of altering subroutines. A brief description of the program follows-

Line number/ subroutine	purpose
1	Identification-the first figure is a chapter reference, the second indicates updates within the chapter.
10	Defines correspondence between machine code and mnemonics. The end of the list is marked by the dummy mnemonic ZZZ.
500-560	Main program
1000	Initialisation
2000	Reads assembly language program into the array A$ until END.
3000	Finds mnemonic code by scanning through the string to find the first group of non-blank characters. Stores the mnemonic code in C$.
4000	Looks for mnemonic code in the table and returns the appropriate machine code for it in C. If a match is not found then GOTO 9000 to report an error.
5000	Gets the address that follows the mnemonic code and stores it in A.
6000	Prints the machine code produced from one line of assembly language and keeps track of the address that it is to be stored in using P.
9000	Prints an error message.

So far the BASIC assembler is not very sophisticated. For example, it doesn't handle errors at all well, but it does contain the start of the subroutines that will be developed to give the full assembler.

If you can't follow the workings of the BASIC assembler so far then don't worry: just type it in and see that it really does change the assembly language in the program swap to machine code.

Storing the code - CLEAR and EXEC

The first part of the BASIC assembler will change mnemonic codes to machine code and produce a list of numbers that corresponds to the assembled program. It has already been pointed out that for a machine code program to be of any use it has to be stored in memory and not just listed on the screen or on paper. There are two stages to storing the program in memory. Firstly, some memory has to be set aside so that BASIC will not try to use it and secondly, each piece of machine code has to be transferred to the correct memory location.

Reserving some memory is easy with the Dragon. The command CLEAR s,h will reserve s memory locations for string storage and move the top of memory down so that memory location h is the highest that will be used by BASIC. In other words CLEAR s,h will reserve memory from address h+1 up to 32767. Once some memory is available, tranferring machine code to it is easy using the BASIC command POKE a,d which will store the number 'd' in the memory location whose address is 'a'. Adding these ideas to the BASIC assembler gives -

```
1 REM BASIC ASSEMBLER V2.2

5 CLEAR 1000,28671


1020 P=28672

6010 POKE P,C
6020 P=P+1
6030 POKE P,A
6040 P=P+1
6050 RETURN
```

Line 5 reserves 1K of memory for machine code programs starting at address 28672 and ending at 32767 This is more than enough for most of the machine code program examples in this book. Line 1020 initialises P to ensure that the first item of machine code is stored in the first reserved memory I location. The changes to subroutine 6000 make it store each item of machine code in a new memory location (in addition to its original job of printing the machine code so that we can examine it.

With these changes, the BASIC assembler now leaves the machine codes stored in the reserved memory when it finishes. If you want to check that this is true then you could use the BASIC command PRINT PEEK(a) to examine the contents of each memory location.

All that is left now is to discover a way of actually running the machine code stored in the reserved memory. Once again Dragon BASIC makes this particularly easy for us in that if provides the EXEC 'address' command which will start the 6809 obeying a machine code program whose first instruction is stored at 'address'. So, to start the swap program running, all you would have to type is EXEC 28672. However, DON'T try this for the swap program because apart from it not doing anything useful (or visible) it doesn't contain any way to stop itself or to return control to BASIC!

ADDing - a simple operation

So far, all that the 6809 inside the Dragon has done is move numbers from memory and back to memory. To show the sort of thing that the A and B accumulators are more often used for and to give an example of a running machine code program, it is worth introducing the new assembly language instructions ADDA 'address' and ADDB 'address'. These simply add the contents of the memory location 'address' to the current contents of the A or B register respectively. Notice that this leaves the contents of 'address' unaltered and the result of the sum stored in the register. This is the reason that the A and B registers are called accumulators because they 'accumulate' the answer to a sum. Suppose that we want to add the contents of memory location 200 to the contents of memory location 201 and, for simplicity, leave the answer in one of the registers. This problem can be solved as follows:

> first, load the A register from one of the locations, then add the contents of the second location to the A register leaving the answer in A

In other words:

```
LDA200
ADDA201
```

Although this is a simple program, notice that it is more complicated than writing C+D to add two numbers in BASIC! It is a characteristic of

assembly language that operations that would be carried out in a single BASIC statement have to be broken down into individual steps. It would be easy to add the mnemonics ADDA and ADDB and their corresponding machine codes 153 and 217 to the DATA statement in the BASIC assembler but it is better to wait until we have looked at a few more 6809 operations and add them all in one go.

Micro projects

1) Using the information given in this chapter, hand assemble the two line addition program in the last section. Show the address that each item of machine code is stored in given that the program is to start at address 28672.

2) If memory location 200 contains 56 and memory location 201 contains 4 before the program is run what do they contain after it is run and what does the A register contain?

Chapter Three

Addressing

There are two parts to any assembly language instruction the operation to be carried out and the address of the memory location that it is to be carried out on. The subject of how assembly language instructions can address memory is very important and in this chapter some of the simpler but more useful methods are described. As addressing is about the use of numbers and how they are stored inside the computer, it is difficult to avoid the subject of binary and hexadecimal numbers. Both of these topics are treated in this chapter, although only in as much as they are useful to the assembly language programmer.

Operations and addresses

You should be able to see that, in all the examples of assembly language instructions that have been introduced so far, there is a general pattern. Each instruction consists of a short mnemonic that determines the operation to be carried out such as LDA or ADDA and a number that determines a memory location that is to be involved in the operation. In general an instruction can take the form -

operation address expression

Where 'operation' is, as already described, a mnemonic defining the operation to be carried out but 'address expression' can be one of the many ways that the 6809 allows a memory location to be specified. At first

it is often difficult to see why a wide range of ways of specifying an address is at all useful! In this chapter only the most obviously useful methods, or 'addressing modes', will be described. The more exotic ones will be dealt with later. So far, the only form of addressing mode that has been introduced is simply writing the address of the memory location after the operation. This is usually known as 'direct addressing' and it is more restricting than has so far been admitted!

Direct Addressing

If you go back and look at the few examples given in the previous chapter you will indeed find that they fit into the format of

operation | address

but what you may not have noticed is that all of the addresses were smaller than 255. This allowed the address to be stored in a single memory location following the machine code for the instruction. For example, LDA 255 can be assembled to 146,255 and each number can be stored in a single memory location but what about LDA 256? Remember, a single memory location can only hold numbers in the range 0 to 255. The answer is of course to use more than one memory location to store the address but this takes us to a second method of addressing memory called 'extended addressing'. In short, direct addressing can be used to specify an address in an instruction only if the address is in the range 0 to 255. Obviously, this is very restrictive and, as you might imagine, direct addressing is not often used. (In fact there is rather more to direct addressing than it is worth going into at this stage and, at a more advanced level, it does have some advantages over extended addressing.) However before we can move on to using extended addressing it is worth looking at some details of how numbers are actually stored in memory.

Bits, bytes and binary

So far all that has been said about the nature of a single memory location is that it can store a number in the range 0 to 255. In fact, a memory location doesn't store a decimal number at all. Instead, it stores a pattern of eight zeros and ones, in other words eight 'bits'. A group of

eight bits forms a single unit, referred to as a 'byte'. What this pattern of eight bits represents is not something that is uniquely defined. For example, you might use each bit to represent the state, open or closed, of eight doors in a house. One of the differences between BASIC and assembly language is the BASIC has numbers and characters to work with but in assembler the only raw material is the bit pattern. It is important to realise that, even though the eight bits that are stored in a memory location are normally interpreted as a number, this is just one interpretation. However, this conventional interpretation is so important it is worth going over the details of standard 'binary numbers'. Later on it will be necessary to change the way that the pattern of bits is interpreted to include negative numbers.

For most people, the decimal system is the best known way of representing numbers. Using the digits 0 to 9 it is easy to count up to nine objects. The answer to how to count beyond 9 is so familiar to all of us that it hardly seems a problem. The decimal system uses a second digit to record the number of groups of ten that have been counted, a third digit for the number of hundreds and so on. This is called a 'place value' system since each place represents a multiple of the 'base' value. In other words, the number 245 is to be understood to mean two lots of 100, four lots of 10 and five lots of 1. The binary system is also a place value system that works in exactly the same way as the decimal system - except that only the figures 0 and 1 are available for counting. This restriction gives rise to a place value system that counts in lots of 1's,2's,4's,8's and so on. So the binary number 101 is to be read from left to right as one lot of 4, no lots of 2 and one lot of 1, giving the number that we call five in decimal. As another example, consider the binary number 1010110. This can be written as:

place value	128	64	32	16	8	4	2	1
	1	0	1	0	1	1	0	0
decimal	128+	0+	32+	0+	8+	4+	0+	0
	=172							

Notice that the place value increases by a factor of 2 for each place to the left and the decimal equivalent is just the sum of the place values wherever a 1 occurs.

It is not too important that you know how to convert binary to decimal and vice versa but it is important that you are not worried by binary

numbers and are prepared to look up the extra details that you need to know. Now the reason why a memory location can hold numbers in the range 0 to 255 should be clear. The smallest number that can be represented by eight bits is 00000000 or 0 and the largest is 11111111 which, if you convert it to decimal, gives 255.

Hexadecimal and binary

The representation of numbers in binary is not difficult to understand but it is tiresome to have to convert from binary to decimal and vice versa. For this and other reasons, assembly language programmers have always tended to use 'hexadecimal numbers' which are very easy to convert to and from binary. Hexadecimal, or just 'hex', numbers also use a place value system but this time there are 16 digits: the usual set of 0 to 9 and six new ones composed of the letters A to F. Because of the use of letters to represent 10 to 15 hex numbers look very intimidating! However, to count in hex is easy, first count 0 to 9 as usual but then instead of counting 10, 11 up to 15 count A, B, C up to F. The hex place value system works in the same way as for binary and decimal except that the value increases by a factor of 16 for each place to the left. For example, the hex number A30F is:

place value	4096	256	16	1
	A	3	0	F
decimal	4096*10 + 256*3 + 0*16 + 15*1 = 41743			

As you can see, the place value increases very rapidly and this means that hex numbers use fewer digits than decimal for the same number.

You don't have to worry about converting hex numbers to decimal and vice versa because the Dragon can do it for you. To convert from decimal to hex use the HEX$ function - e.g. PRINT HEX$(41743) - and to convert a hex number to decimal simply precede it by &H - e.g. PRINT &HA30F. In fact the Dragon can do a little better than just converting hex numbers to decimal because you can use a hex number anywhere that you can use an ordinary number.

Unfortunately, the Dragon offers us no help in converting from hex to binary and vice versa. However, this is not difficult. It takes four bits to represent a number in the range 0 to 15 (to check, convert 1111 to decimal) and so a single hex digit can be represented by four bits.

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

To convert from hex to binary all you have to do is take each hex digit in turn and write down its four bit binary equivalent from the table. For example the binary equivalent of F3A2 is -

F	3	A	2
1111	0011	1010	0010

To convert from binary to hex all you have to do is form the bits into groups of four starting from the right then look up each group in the above table. For example, to convert the number 101101110 to hex -

0001 |	0110 |	1110
1	6	D

Perhaps the best reason for using hex numbers is the simple observation that a two figure hex number corresponds to eight bits and so can be stored in a single memory location. In other words a single memory

location can store a hex number in the range 00 to FF. In the same way a four figure hex number can be stored in exactly two memory locations and so on. It is this correspondence between the number of figures in a hex number and the number of memory locations it takes to store it, coupled with the natural way hex and binary work together, that makes hex numbers so useful in assembly language programming. From now on, unless there is a good reason otherwise, all of the numbers used in this book will be hex numbers. Although the Dragon uses &H in front of a number to indicate that it is in hex it is almost universal in assembly language to use $ in front of a hex number and this will be used in the rest of this book. So 10 is ten but $10 is 16!

Extended addressing

After a long digression into binary and hex numbers it is time to return to 6809 addressing modes. Direct addressing uses a single memory location to store the address of the memory location to be used in the operation. Extended addressing takes two memory locations to store the address. This corresponds to an address in the range 0 to 65535 which is as much memory as the 6809 can handle. Thus using extended addressing you can select ANY memory location within the Dragon and there is no need to extend the number of memory locations used to store an address to three or more.

To distinguish between an instruction with a direct and an extended address is fairly easy:

```
    LDA$3F
is a direct address because it is less than $FF but
    LDA$3F2C
needs two memory locations for the address and so it is extended.
```

What is more interesting is that the machine code for the two instructions is different. The code for LDA with direct addressing is $96 and its code using extended addressing is $B6. (Notice the use of hex numbers to represent the machine codes.) Notice that this means that it is no longer possible to convert assembly language mnemonics to machine code simply by looking them up in a table. Now we have to take into account the addressing mode used as well as the mnemonic. If you look at the row

containing the mnemonic LDA in the table of Appendix I, you will see that there is a range of machine codes, each one corresponding to LDA used with a different addressing mode. You should be able to find the two values quoted above in the columns headed 'direct' and 'extended'. From now on, the BASIC assembler is going to have to examine both the mnemonic and the addressing mode before it can determine the machine code for a given instruction. So

```
LDA $3F2C
```

hand assembles to

```
B6 3F 2C
```

Where each pair of numbers occupies a single memory location. Notice that if the instruction code $B6 is stored at memory location 'a' then the first pair of figures of the address are stored at 'a + 1' and the second pair at 'a + 2'. In other words, the 'most significant' byte (eight bits) is stored first and the 'least significant' byte next.

A practical program

Now that we can address all of the 6809's memory it is possible to write a program that will add the contents of any two memory locations together. In other words, the program will be the assembly language equivalent of the one line BASIC program -

```
10 A=B+C
```

First, it is important to notice that, unlike the BASIC program, all that the assembly language program can do is to add two numbers in the range 0 to $FF. For the sake of simplicity let's use memory location $7FFF to store the result and $7FFD and $7FFE to store the two numbers to be added together (see fig 3.1). ($7FFF is 32767 which is the highest Dragon address occupied by RAM used for storing programs.) The program has to perform the assembly language equivalent of:

```
add the contents of $7FFD to $7FFE
and put the result in $7FFF
```

As already explained at the end of Chapter Two you cannot implement this idea as it stands in assembly language. The reason for this is that the only add instructions that the 6809 has add the contents of a memory location

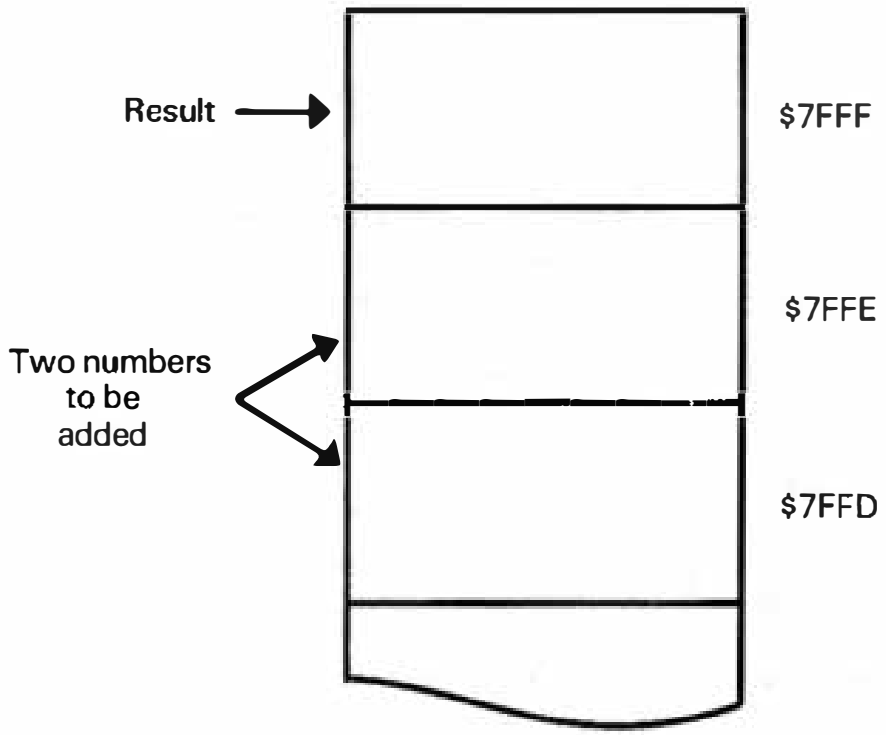


Fig 3.1 Data for simple addition

to the A or B registers, where they also leave the result. So the program has to written

```
LDA$7FFD
ADDA$7FFE
STA$7FFF
RTS
```

You should recognise this as being essentially the same as the addition program given at the end of Chapter Two but now using extended addressing. The instruction RTS has been added to the end of the program so that when it is finished the program returns to BASIC rather than running on out of control. The details of RTS will be discussed later but, for now, all that you have to know is that assembly language program that you intend to run should always end with RTS.

Once again it is worth hand assembling the program to see what the BASIC assembler has to do. The only real difference is that now, when looking up the mnemonic codes in Appendix I, you have to take into account the addressing mode used. In this case the extended mode is used throughout, apart from RTS for which there is only one choice of machine code anyway! If the program is loaded into memory starting at $7000 (which is the same as 28672 used in Chapter Two) then the finished machine code program is -

```
7000 B67FFD
7003 BB7FFE
7006 B77FFF
700939
```

Notice that the listing is given totally in hex but the usual $ sign signifying a hex number has been left out to make the listing more readable. Also notice how much neater the listing is for being in hex: there are always four figures to the address in the first column, every pair of hex figures is stored in its own memory location, with each pair stored consecutively. Thus B6 is stored in memory location 7000, 7F at 7001, FD at 7002 and so on to 39 stored at 7009. The time has now come to modify the BASIC assembler so that it can handle different addressing modes and at last run our first assembly language program

Adding addressing modes to the BASIC assembler

The main change to the BASIC assembler from V2.1 is that now the DATA statements holding the information on each mnemonic code have to include the machine code for each addressing mode possible on the 6809. Rather than just making allowance for the two addressing modes, direct and extended that have been introduced so far it makes sense to write DATA statements for each of the five possible addressing modes. A suitable format and the one that will be used for the rest of the program is

DATA mnemonic,code1,code2,code3,code4,code5

Where code1 through code5 are the machine codes for the five addressing modes 1 to 5. The names of these modes are -

1 Immediate
2 Direct
3 Indexed
4 Extended
5 Inherent

The only problem is that any given mnemonic may not use all five addressing modes. To record the fact that any addressing mode is illegal with the mnemonic in question a code of -1, which normally cannot occur is used. Once the format of the DATA statements has been fixed the rest of the assembler follows. -

```
1 REM BASIC ASSEMBLER V3.1

5 CLEAR 1000,&$6FFF

10 DATA LDA,&H86,&H96,&HA6,&HB6,-1
11 DATA LDB,&HC6,&HD6,&HE6,&HF6,-1
12 DATA STA,-1,&H97,&HA7,&HB7,-1
13 DATA STB,-1,&HD7,&HE7,&HF7,-1
14 DATA ADDA,&H8B,&H9B,&HAB,&HBB,-1
15 DATA ADDB,&HCB,&HDB,&HEB,&HFB,-1
16 DATA RTS,-1,-1,-1,-1,&H39
99 DATA ZZZ,-1,-1,-1,-1,-1

500 GOSUB 1000
510 GOSUB 2000
```

```
520 GOSUB 3000
530 GOSUB 4000
540 GOSUB 5000
550 GOSUB 6000
560 I=I+1
570 IF I> T THEN STOP
580 GOTO 520

1000 DIM A$(50),C(5)
1010 I=0
1020 P=&H7000
1030 RETURN

2000 INPUT L$
2010 IF L$="END" THEN T=I:I=1: RETURN
2020 I=I+1
2030 A$(I)=L$
2040 GOTO 2000
3000 J=1
3010 IF MID$(A$(I),J,1)=" " THEN J=J+1:GOTO 3010
3020 M$=MID$(A$(I),J,1)
3030 J=J+1
3040 IF J<=LEN(A$(I)) THEN IF MID$(A$(I),J,1)<>" "THEN
M$=M$+MID$(A$(I),J,1):J=J+1:GOTO3040
3050 J=J+1
3060 RETURN

4000 RESTORE
4010 READ C$
4015 FOR K=1 TO 5:READ C(K):NEXT K
4020 IF C$="ZZZ" THEN I=I+1:ER=1:GOTO 9000
4030 IF C$=M$ THEN RETURN
4040 GOTO 4010

5000 GOSUB 5500
5010 IF AF$="" THEN TYPE=5:RETURN
5020 A=VAL(AF$)
5030 IF A<256 THEN TYPE=2
5040 IF A>=256 THEN TYPE=4
5050 RETURN
```

```
5500 AF$ = ""
5510 FOR K = J TO LEN(A$(I))
5520 L$ = MID$(A$(I),K,1)
5525 IF L$ = "$" THEN L$ = "&H"
5530 IF L$< >" " THEN AF$ = AF$ + L$
5540 NEXT K
5550 RETURN

6000 PRINT HEX$(P);TAB(5);HEX$(C(TYPE));TAB(10);
6010 POKE P,C(TYPE)
6020 P = P+ 1
6025 IF TYPE=5 THEN PRINT:RETURN
6030 IF TYPE=2 THEN PRINT HEX$(A):POKE P,A
6040 IF TYPE=4 THEN PRINT HEX$(A):POKE
P,INT(A/256):P = P + 1:POKE P,A-INT(A/256)*256
6050 P = P + 1
6060 RETURN

9000 PRINT "ERROR --";ER;"***"

9010 RETURN
```

In going to this second version, the opportunity has been taken to change all of the relevant constants to hex and to change the listing of the program to hex. The main changes are in subroutines 4000, 5000 and 6000. Subroutine 4000 now scans through the DATA statements to find the mnemonic stored in M$ but now, when it finds it, it returns the five possible codes in the array C rather than a single code. Subroutine 5000 now has not only to work out the address following the mnemonic but has to determine the type of the address and so select the correct code from the array C. A new subroutine has been added (5500) which called at the start of subroutine 5000. Subroutine 5500 takes the addressing information and packs it character by character into the variable AF$ (AF standing for Address Field). As this transfer is accomplished, a number of changes are made. First, any blanks that have been included are removed and, secondly, any $ signs are changed to &H so that the BASIC VAL function can work the address out correctly. Subroutine 5000 then sets the variable TYPE to 2 if the address is less than 256 and to 4 if it is greater than 255. If there is no address field i.e. if AF$ = "" then the instruction must be like RTS and then the appropriate type is 'inherent' or TYPE=5. Subroutine 6000 is changed to PRINT and POKE a different number of items depending on the addressing mode.

Notice that only DATA statements for the 6809 instructions that have been described so far have been included in the program. In principle there is no reason why the list of DATA statements shouldn't be extended to include all of the instructions in the operation code table given in Appendix I. However there are a great many instructions in the table and typing in all the DATA statements in one go is no small task! If you don't feel like tackling this particular project then the easier alternative is to add the appropriate DATA statements as instructions are encountered in future chapters. An advantage of this method is that the BASIC assembler grows as your knowledge does!

Trying it out

After typing in the above version of the BASIC assembler the obvious thing to so is to try to assemble and run the simple addition program. RUN the assembler and type in

```
LDA$7FFD
ADDA$7FFE
STA$7FFF
RTS
END
```

You should see the same listing that you got from the earlier hand assembly. After the assembler has finished the machine code is stored in memory starting at $7000. Once again you can check that this is so by PEEKing each of the memory locations i.e. PRINT HEX$(PEEK(&H7000)) etc. You can run the machine code by simply typing EXEC &H7000. If everything goes according to plan (or rather program) you should see the usual Dragon OK printed on the screen indicating that the machine code program has returned control to BASIC. If anything else happens your only hope of regaining control of the machine is to press the reset button. If you are lucky your program will be intact, on the other hand it might not be! It is worth learning very early on that assembly language is not as friendly or as safe as BASIC. If you make a mistake in a machine code program you won't get a error message, just a Dragon that behaves oddly.

If everything did work OK, and with such a short program there is no reason why it shouldn't, then you should find the the content of memory location $7FFF is indeed the sum of the contents of $7FFD and $7FFE. The easiest way to check that this is true is -

```
PRINT PEEK(&H7FFF),PEEK(&H7FFD),PEEK(&H7FFE)
```

You can use POKE to change the contents of the two memory locations $7FFD and $7FFE and then EXEC the machine code again just to make sure that it all works.

Immediate addressing

So far, the only two addressing modes that we have looked at - direct and extended - are hardly different enough to give the flavour of the subject. After extended addressing, the most frequently used addressing mode is probably 'immediate addressing' and so it is important to introduce this mode as early as possible. Consider the problem of adding a constant, for example, 3 to the contents of a memory location. So far, the only way that this could be done would be by using the addition program used as an example in this chapter. Immediate addressing is sometimes called 'immediate data' because it allows the data to be stored in the memory location following the machine code for the instruction. Notice that in this case the number following the instruction code IS the data not the address of the data. In assembly language, immediate addressing is indicated by a # sign. For example,

```
LDA #$3
```

loads the A register with 3 in contrast to

```
LDA $3
```

which loads the A register from memory location 3. The hand assembly of LDA #3 is -

```
86 03
```

To change the BASIC assembler to handle immediate addressing is easy. Simply add the following lines -

```
1 REM BASIC ASSEMBLER V3.2

5025 IF TYPE=1 THEN RETURN
```

```
5522 IF L$ = "#" THEN TYPE= 1:AF$ = "":GOTO 5540

6045 IF TYPE = 1 THEN PRINT HEX$(A):POKE P,A

6060 TYPE = 0
6070 RETURN
```

Notice that immediate addressing doesn't make sense with all types of instruction. For example, STA #3 is illegal because there would be no point in storing the contents of the A register in the memory location following the instruction code so what would the 3 mean? Immediate addressing only makes sense when used with instructions that obtain data from memory and not with those that store data in memory.

Summary

1) There are many ways of specifying where the data that an instruction will operate on is located in memory. These ways are called 'addressing modes'.

2) Direct addressing is used to access data in memory locations0 to $FF. In this case, the address is stored in a single memory location following the instruction code.

3) Extended addressing can address memory from 0 to $FFFF which is the whole of the6809's address range. In this case, the address is stored in the two memory locations following the instruction code.

4) Immediate addressing is different from the previous two modes in that the data for the instruction is stored in the memory location following the instruction code.

5) Bit patterns and binary numbers are the fundamental data of assembly language programming.

6) Hex numbers are more convenient then decimal numbers and are almost always the standard way of writing addresses, data and instruction codes in assembler.

Micro Projects

1) How many memory locations does the number $F3095E6F require to be stored in memory?

2) Convert the following hex numbers to binary -

a) $0100
b) $1000
c) $7FFF
d) $FFFF

Do any of these numbers seem familiar!?

3) What is wrong with -

a)STB #$34
b) LDA #$FF32

4) Hand assemble the following program -

```
LDA #$10
ADDA$7FFF
STA $7FFF
RTS
```

Check your answer by using the BASIC assembler.

What does this program do?

Chapter Four

Jumps, Loops and Labels

Imagine how limited BASIC would be if you didn't know about the GOTO statement. Without GOTO, you could not code jumps or loops in your program. This is such a fundamental part of programming in any language that it is important that you become familiar with the assembly language equivalent of GOTO. This also leads us on to consider what is really the last new idea to be included in the BASIC assembler - labels. If you have been disappointed by the example assembly language programs in the earlier chapters because of how little they manage to do, then you will be a little happier with the main example in this chapter, at least it displays something on the screen!

The JMP instruction

As described in Chapter Two, the address of the memory location that an instruction is stored in is in some ways like the line number of a BASIC command. In particular the assembly language instruction -

JMP address

will cause the next instruction to to be the one starting at 'address'. There are three addressing modes that can be used with the JMP instruction, direct, extended and indexed, the last of which is yet to be described. So JMP $4323 is a valid JMP instruction which causes the instruction at $4323 to be carried out next. If there isn't an instruction stored at this address then things will go very wrong. Once again, unlike BASIC, machine code won't give you a friendly error message to tell you that there is no such line number, rather address, because of course the address referred to in a JMP instruction always exists! What happens if you JMP to a memory

location that doesn't contain machine code is that the 6809 will obey whatever collection of numbers the memory does contain as machine code and so the result is entirely unpredictable behaviour. For this reason you have to be very careful that JMP instructions do go to the correct location.

The JMP instruction look simple enough to use for any one familiar with the GOTO instruction. However there is fundamental problem that both GOTO and JMP share. If you are in the middle of writing a BASIC program and you know that you want to GOTO a line further on in the program then how do you know its line number? The trouble is that you only know the line numbers of commands that you have already written. This is known as the 'forward jump' problem (see fig 4.1). The usual solution in BASIC is to either guess the line number and then correct the GOTO statement when the program is finished or just leave it undefined and go back and fill in the missing line number later. In assembly languagge the problem is agravated because while you are writing the program using mnemonics you don't know the address that any given instruction will be stored at, this is something that is only discovered when you assemble (by whatever method) the program. This makes 'backward' jumps difficult to handle, let alone forward jumps! For example, suppose that you are writing a program that will load the A register from a number of memory locations in turn and store its contents in a single location (why you might want to do this will become clear after the sound producing examples in Chapter Nine.) The program that you would write might look something like -

```
LDA$7F00
STA$FF20
LDA$7F01
STA$FF20
JMP????
```

Where the four question marks indicate that the address for the JMP is unknown. To make the program into a loop the JMP should transfer control back to the first instruction of the program. The address of the first instruction obviously depends on where the program is loaded into memory. If the program is loaded, as in all the previous examples, starting at $7000 then obviously the correct address for the JMP instruction is JMP $7000. However, suppose you wanted to JMP to the third instruction in

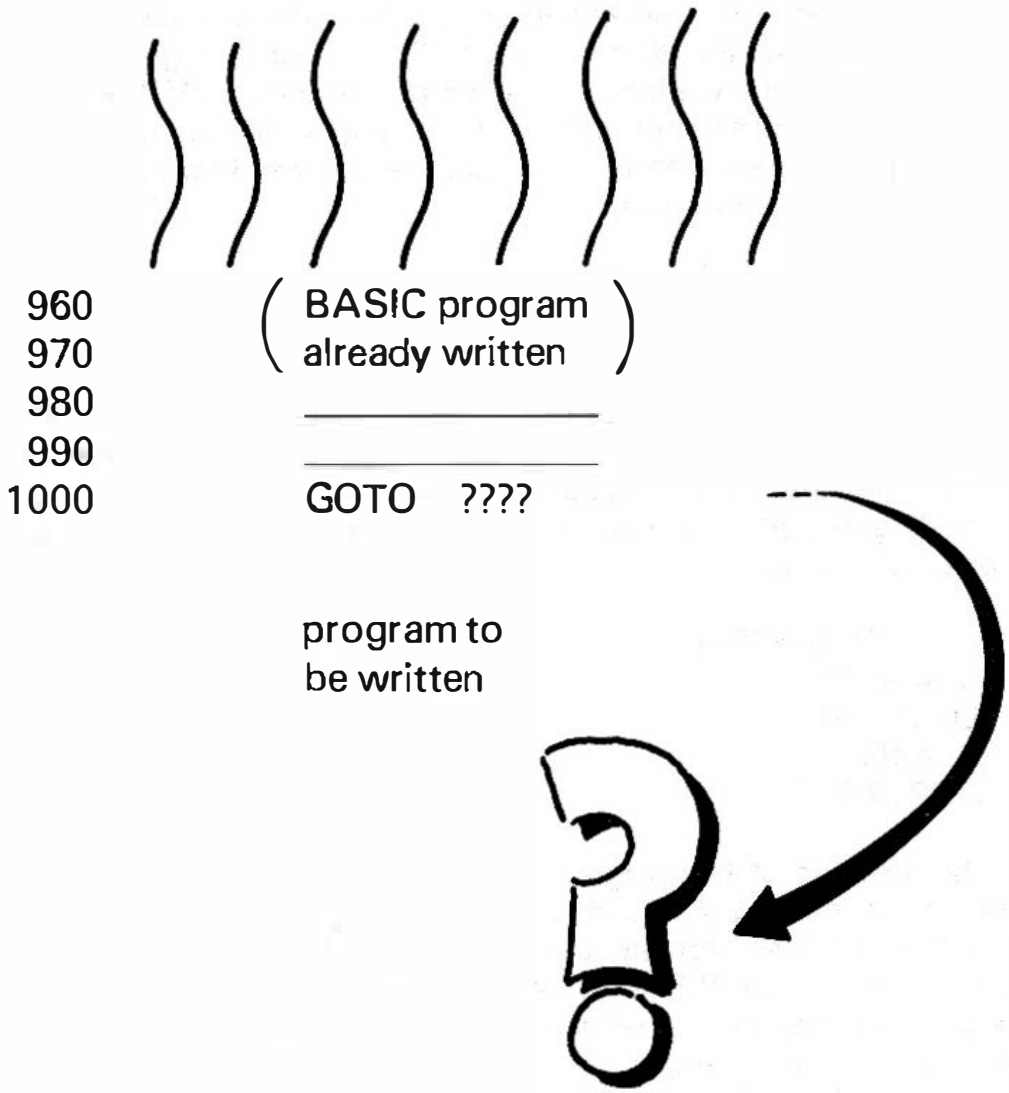


Fig 4.1 The forward jump problem

the program, i.e. LDA $7F01, how would you know its address? The answer is that you would have to assemble the program! This doesn't sound too difficult until you realise that if you make any changes to the program that inserts or deletes instructions then the chances are that the address of the instruction that you are JMPing to will change and, to make the program continue working, you have to change the JMP addresses. If you think about the difficulties of actually using an instruction that jumps to a particular address then you will soon see the need for adding address labels to the BASIC assembler.

Address labels

The use of a mnemonic code to represent the machine code corresponding to an instruction should seem like a fairly straightforward but very useful idea by this stage. A very similar idea can be used to make the handling of addresses just as easy. If the example of the last section had been written as -

```
START LDA $7FF0
STA$FF20
LDA$7FF1
STA$FF20
JMPSTART
```

then the meaning of the program would have been clear. The final JMP instruction is obviously intended to transfer control to the instruction 'labelled' START so forming a loop. When the program is assembled START as used in JMP START has to be changed to an address. To be more precise it has to be changed to the address of the instruction that it labels. START is an example of an 'address label' (or just 'label') There are just enough similarities between a BASIC variable and a label to be confusing. A BASIC variable has a value associated with it and so does a label (i.e. the address of the instruction that it labels) but a label is used only in the translation of assembly language to machine code whereas a BASIC variable exists while a program is running. It is better to think of a label as more like a mnemonic code that stands for an address rather than as one which stands for machine code.

There are two ways that a label can appear within a program:

1) As part of an instruction, standing in place of an address.

2) In front of an instruction, so labelling a position within the program.

The use of a label in front of an instruction can be thought of as defining its value or address. If you think about it for a moment, it only makes sense to define a label once in a program but once so defined it can be used as part of any instruction as often as required.

Adding Labels to the BASIC assembler

Adding a label facility to the BASIC assembler is very easy. First, a new pair of arrays needs to be defined, T$ to hold the labels and T to hold their corresponding address values. When a label is defined, T$ is searched and as long as the label isn't already in T$ (in which case an error is reported) it is added to the list and its address is stored in T. The only problem is, how do we tell the difference between a label and an ordinary mnemonic code? Some assemblers demand that a label should always start at the beginning (i.e. without any blanks before it) and a mnemonic should always have at least one blank before it. Following the convention of DASM, the assembler from Compusense, labels will be distinguished by starting with "@". So legal labels take the form:

```
    @START
    @LOOP
etc.
```

This convention is very handy because it allows the BASIC assembler to detect a label by examining just the first character. If, when using this method, a label is detected following the mnemonic code, then the array T$ is searched. If the label isn't found, then it hasn't been defined and an error should be reported. If the label is present in T$ then the

corresponding address stored in T is used to replace the usual contents of AF$. The changes required are -

```
1 REM BASIC ASSEMBLER V4.1

17 DATA JMP,-1,&H0E,&H6E,&7E,-1

1000 DIM A$(50),C(5),T$(50),T(50)
1030 LC=0

1400 RETURN

3060 IF LEFT$(M$,1)="@" THEN GOTO 3500
3070 RETURN

3500 S$=M$
3510 GOSUB 7000
3520 IF F>0 THEN ERR=0:GOTO 9000
3530 LC=LC+1
3540 T$(LC)=M$
3550 T(LC)=P
3560 GOTO 3010

5550 IF LEFT$(AF$,1)<>"@" THEN RETURN
5560 S$=AF$
5570 GOSUB 7000
5580 IF F=0 THEN ERR=3:GOTO 9000
5590 AF$=STR$(T(F))
5600 RETURN


7000 K=1
7010 IF K>LC THEN F=0:RETURN
7020 IF T$(K)=S$ THEN F=K:RETURN
7030 K=K+1
7040 GOTO 7010
```

Apart from the updates to the existing subroutines, this version includes two new modules. Lines 3500 to 3560 check for the existence of the label in the array T$ and, if it isn't there, it adds it along with the current address in T. Subroutine 7000 searches for the string S$ in the array T$. On, return

the variable F is 0 if the string wasn't found and equal to its position in the array if it is. Lines 5550 to 5600 detect a label in the address field, use subroutine 7000 to find it in the array T$ and then substitute the address stored in T. Also notice that the DATA statement for the JMP instruction has been included.

To test the new version of the BASIC assembler try -

```
@STARTLDA$7FF0
STA$FF20
LDA$7FF1
STA$FF20
JMP@START
```

The ability to handle labels can be seen in the way that the last line is assembled to -

```
7E 7000
```

Now that labels can be used, the JMP instruction is easy to use. All you have to do is label any instruction in the program that you want to JMP to and, as long as you don't define the same label twice, you can JMP to it as often as you like. However, this simple picture is still spoiled by the problem of forward jumps.

Forward jumps and two-pass assembly

The BASIC assembler V4.1 will handle labels but only if they are defined before they are first used in an address field. For example, if you try to assemble -

```
LDA$7FF0
JMP@SKIP
STA$FF20
@SKIPLDA$7FF1
END
```

you will get an error message because when the assembler reaches the JMP @SKIP instruction the label @SKIP is not yet defined - it is further down the program. This restriction on the use of labels is so serious that, if allowed to persist, it would make assembly language programming very difficult. Fortunately the solution is quite simple. To make sure that any label that might be used in a program is defined, all that the assembler has to do is read through the program, picking up all the label definitions before it attempts to produce a final machine code assembly. In other words, the assembly should make two passes though the assembly language program. The first pass just serves to record the label definitions and the second pass uses these definitions to produce a correctly assembled program. Notice that the first pass that the assembler makes through the program has to assemble the program the best it can so that the instructions take up the correct amount of memory and the labels are defined at their correct addresses. This implies that the easiest way to change the V4.1 assembler into a two-pass assembler is just to make it automatically run twice over any assembly language program, the first time ignoring any errors that are generated because labels are undefined and the second time using the definitions collected in the first pass.

Changing the BASIC assembler into a two-pass assembler

Adding a two-pass facility to the BASIC assembler is much simpler than you might think. All that has to be done is to change the main program into a loop,

```
initialise 1
FOR PASS = 1 TO 2
initialise 2
assemble
NEXT PASS
```

and be careful about where things are initialised. For example, there would be no point in doing the first pass if the array T$ used to hold the labels was cleared between passes! The only other changes to the assembler involve the use of the variable PASS to decide when an error should be reported. If an undefined label is encountered then this is now only an

error if the assembler is on pass 2. Also, as all the labels should have been defined after pass 1, discovering that a label is already in T$ isn't an error in pass 2. Make the following modifications to the BASIC assembler:

```
1  REM BASIC ASSEMBLER V4.2

515 FOR PASS=1 TO 2
518 I=1:P=&H7000

570 IF I<=T THEN GOTO 520
575 PRINT
580 NEXTPASS
590 STOP

3520 IF F>0 AND PASS=1 THEN ERR=2:GOTO 9000
3525 IF F>0 AND PASS=2 THEN GOTO 3010

5580 IF F=0 AND PASS=2 THEN ERR=3:GOTO 9000
```

If you run the new version of the assembler on any of the previous examples you will find that you now get a listing of the machine code twice - once on each pass through the program. Although the second one is the only one that has any chance of being correct, the first listing does provide some information about the way that the assembler is working and so it will be left as a feature for the time being, until a better version of subroutine 6000 is written later on.

When an undefined label is encountered on the first pass the value zero is used as its temporary address so that the rest of the program can be assembled. However, the current way that the assembler selects direct or extended addressing causes something of a problem. For example, if you try -

```
JMP @SKIP
LDA #0
@SKIPLDA #1
```

(a correct program that doesn't do anything useful!) you will find that on the first pass the JMP @SKIP is assembled as a direct address because @SKIP has the temporary address of 0 but on the second pass JMP

@SKIP is assembled as an extended address because @SKIP now has an address greater than 255. This change in addressing modes is a problem because extended addressing uses one more memory location than direct and so, on the first pass, the address that @SKIP labels is different from the address it labels on the second pass - obviously, the resulting machine code will not work. The problem is that on the first pass the assembler doesn't know whether undefined labels are going to be defined as direct or extended addresses. The solution to the problem is to change the way that a direct address is selected as the correct addressing mode. If the assembler assumes that, unless otherwise informed, all addresses are extended then the amount of memory used on the first pass will be the same as the amount of memory used on the second pass and the addresses assigned to labels will not change. After all, there is no real need to use direct addressing, other than to save one memory location. In other words LDA $0030 (in extended mode) works just as well as LDA $30 (in direct mode). Later on, other features of direct addressing will be explained that make it more useful so, to be able to carry on using it, the convention that a direct address must start with a '>' sign will be used from now on. So

```
LDA >$30
```

will be assembled as a direct address but

```
LDA $30
```

will be assembled as an extended address.

The changes to the BASIC assembler to make extended addressing the default mode and introduce the new symbol for direct addressing are:

```
5030 IFTYPE=2 THENRETURN
5040 TYPE=4

5521 IF L$=">" THEN AF$="":TYPE=2:GOTO 5540
```

Now the BASIC assembler will handle forward and backward JMPs by making two passes through the program correctly. To check this, try assembling any of the previous examples. Make sure that the address following a JMP instruction really is the address of the first memory location in which the correct instruction is stored.

Subroutines, JSR and RTS

Now that the assembly language equivalent of the BASIC GOTO has been introduced, it is only a small step to the assembly language equivalents of the BASIC GOSUB and RETURN. The JSR address (Jump to SubRoutine) will transfer control to the instruction stored at 'address' but like the BASIC GOSUB it causes the 6809 to store the address of the instruction following it. This stored address is used by the instruction RTS (ReTurn from Subroutine) to transfer control back after the JSR. Thus the JSR and RTS pair exactly mimic the behaviour of the BASIC GOSUB and RETURN instructions and so allow us to write and use assembly language subroutines.

The JSR instruction can be used with the same three addressing modes as the JMP instruction - direct, extended and indexed. The fact that the RTS instruction is a little odd when it comes to addressing modes has already been mentioned. The RTS instruction does use an address, but the address is supplied by the 6809. In other words, when you use the RTS instruction, you do not have to explictly give the return address and in this sense the address is 'inherent' in the instruction. There are other instructions like RTS that sometimes do not need a programmer-supplied address and this gives us a fourth addressing mode to add to direct, extended and immediate: 'inherent addressing'. (Corresponding to TYPE=5 in the BASIC assembler.)

Before going on, enter the DATA statement for the JSR instruction in the BASIC assembler (RTS is already present):

```
18 DATA -1,&H9D,&HAD,&HBD,-1
```

Using subroutines - more labels

Since the JSR instruction uses the same addressing modes as the JMP instruction it make sense to allow labels to be used in the same way. However, the JSR instruction is often used to gain access to machine code subroutines already present in the BASIC ROM and this makes the use of labels a little more complicated. For example, there is a machine code subroutine that will print a character on the Dragon's text screen that starts at $800C. The character to be printed is stored as its ASCII code in

the A register before transferring to the subroutine. The following short program will repeatedly print the letter A on the screen -

```
@LOOP LDA #$41
JSR $800C
JMP @LOOP
END
```

The first instruction loads the A register with $41 which is the ASCII code for the letter A. The second instruction calls the subroutine at $800C which prints the A on the screen and then returns control to the JMP instruction which repeats the program forever - or until you press the reset button. If you enter this program using the BASIC assembler you can run it using the usual EXEC &H7000. After the screen has filled with As you may find it difficult to see the As being printed - this is your first taste of how much faster assembly language is than BASIC!

Programs would be easier to read and assembly language generally more friendly if labels could be defined to correspond to fixed addresses outside (i.e. in the ROM) the program that you are currently writing. For example, if the label @PRINT could be defined to be $800C the JSR instruction in the previous example could be written as JSR @PRINT which is much easier to understand. Most assemblers allow a label such as @PRINT to be defined by the statement -

```
@PRINT EQU address
```

where address is a number in the range 0 to $FFFF. The statement would be read as 'the label @PRINT is EQUal to address' and its effect would indeed be to set @PRINT to correspond to the 'address'. The form of the statement looks like a standard assembly language instruction but as should be obvious that EQU ISN'T an assembly language mnemonic. It is an instruction to the ASSEMBLER to give a label a value to be used in the rest of the assembly. Instructions like EQU are called 'pseudo operations' because they look like 6809 mnemonics but they are in fact instructions to the assembler. The pseudo op EQU is worth adding to the BASIC assembler.

Adding EQU to the BASIC assembler

EQU is just the first of a number of pseudo ops to be incorporated into the BASIC assembler so it makes sense to prepare for them by adding EQU in a general way.

```
1 REM BASIC ASSEMBLER V4.3

550 IF PS=0 THEN GOSUB 6000
555 IF PS>0 THEN GOSUB 6500
560 I=I+1:PS=0

4001 IF M$="EQU" THEN PS=1:RETURN

6500 IF PS<>1 THEN RETURN
6510 IF PASS=1 THEN T(LC)=A
6520 IF PASS=2 THEN PRINT TAB(15);A$(I)
6530 RETURN
```

As the form of the EQU statement is the same as a standard assembly language instruction, the approach used is to let the assembler process it as usual and then correct the result later on. Line 4001 detects the EQU and sets PS (for PSeudo op) to 1. The label in the instruction is automatically added to the array T$ but its definition is the address of the current instruction rather than the address following the EQU. This is corrected by subroutine 6500 which also handles the printing of pseudo ops on the second pass.

Using EQU

Now that the BASIC assembler can define labels using the EQU pseudo op, the previous 'print A on the screen' program can be written as-

```
@PRINT EQU $800C
@LOOP LDA #$41
JSR @PRINT
JMP@LOOP
END
```

Once again you should be able to enter the program to the BASIC assembler and then run it using EXEC &H7000 with the same result as before.

The interesting thing about EQU is that, once you have introduced it, there are all sorts of other ways that you can use labels. For example, the second instruction in the program loads the A register with the ASCII code for the letter 'A' but this is not something that is very clear just by reading the program. If however the program is written as -

```
@PRINT EQU $800C
@A EQU $41
@LOOP LDA #@A
JSR @PRINT
JMP @LOOP
```

then the use of a label as part of the immediate address field of the LDA instruction once again makes the program slightly more readable. What might surprise you is that this program can be assembled by the current version of the BASIC assembler without any modifications! In other words, although not their primary purpose, labels can be used in place of data values in instructions.

As a final example of using the @PRINT subroutine try -

```
@PRINT EQU $800C
LDA #0
@LOOP JSR @PRINT
ADDA #1
JMP @LOOP
END
```

The first instruction loads the A register with 0. The next three instructions form a loop that prints the character corresponding to the ASCII code in the A register, then adds one to the A register and so on. The result is that every character that the Dragon can display is repeatedly printed on the screen. You may be wondering what happens when the contents of the A register reach 255 and you try to add one more to it? The answer is that it resets to zero and so the whole cycle repeats itself.

Summary

1) The main topics of this chapter have been the way the instructions JMP, JSR and RTS are used to transfer control together with the

idea of address labels and their implementation. Inherent addressing was introduced.

2) The problems encountered with forward jumps lead to the solution of a two-pass assembler, the first pass to gather the definitions of the address labels and the second pass to correctly assemble the program.

3) Finally the idea of an assembler pseudo op was introduced by way of EQU used to set labels to given values.

Micro Projects

1) Add subroutine 6900 to print a list of all the labels used in a program along with their corresponding addresses. Make sure that the table is only printed at the very end of the second pass.

2) A machine code subroutine starting at $8006 in the BASIC ROM reads the Dragon's keyboard and returns the result in the A register. If no key is pressed then the A register contains 0. If a key is pressed then the A register contains the ASCII code of the corresponding character. Write and test a machine code program that will read the keyboard and print the result on the screen no matter what it is. Try to make an intelligent use of labels in your program.

Chapter Five

Logic Instructions

At some point in learning assembly language, you have to become acquainted with the entire range of instructions that are at your disposal. This problem didn't arise when you were learning BASIC because the operations of BASIC are the familiar operations of arithmetic, +,-,* and /. In assembler the only type of data that you have is a pattern of bits and each of the operations is concerned with changing or manipulating these bit patterns. The problem for most people beginning assembly language is to see how to use these primitive operations to produce meaningful operations on the data that the bit patterns represent.

This may sound like a very abstract and strange way of thinking about things. Surely a bit pattern is always nothing more than a binary number? The answer is, that while a bit pattern can always be interpreted as a binary number, this is not always the best way to think about it. For example, if you consider the contents of two memory locations as numbers in the range 0 to $FF then it makes sense to consider arithmetic operations, such as addition, on these numbers. However, suppose the bit patterns stored in the two memory locations are being used to represent characters, i.e. they are ASCII codes, then, although they can still be interpreted as numbers, adding them together makes very little sense!

This lack of familiarity with the use of bit patterns doesn't make the range of machine code operations any more difficult to understand; after all, they are very simple operations. What is difficult is seeing how these very simple operations can be put together to do anything useful. The purpose of this and the next chapter is to explain the action of all of the 6809's operations on data. At this stage you may find that you understand what the operations are doing, but not why you would ever want to use some of them. For now, concentrate on gaining a rough idea of what the

operations are all about and later on as you examine the examples and write your own programs you will find that the use of each instruction will become clear.

The logical operations

The logical operators AND, OR and NOT should be familiar to you from Dragon BASIC and their use in IF statements. What you might not be so familiar with is that AND, OR and NOT are operators in the same sense as +,* etc are. The only difference is that, instead of working on numbers, AND, OR and NOT operate on the two values True and False, or T and F for short. If A and B are two statements that are either true or false then the following table illustrates the meaning of A AND B

A	B	A AND B
F	F	F
F	T	F
T	F	F
T	T	T

You should be able to work out the result of each line of the table using nothing more than common sense. For example, if A is false and B is true then the combined statement A AND B is clearly false. In fact the statement A AND B is only true if BOTH the statements A and B are true.

You probably already know that the above table is called a 'truth table' and that all of the operations of logic can be summarised by similar truth tables. The way that the logical operators of 6809 assembly language are also best described by using truth tables. The only real change is that instead of T and F they work in terms of 1 and 0 and on all the bits of a memory location at once.

ANDA address and **ANDB** address

The 6809's AND, instruction in keeping with all of the other logical instructions, will AND the contents of a memory location with the current contents of the A or B register where it also leaves the result. The idea that

an operation works on a value from memory and a value in one of A or B is something that is familiar from the way that ADDA and ADDB work. The truth table for the AND operation is -

reg.	mem.	result
0	0	0
0	1	0
1	0	0
1	1	1

where reg. and mem. signify a bit in the appropriate register and in memory respectively. The only extra complication is that the operation is applied 'bitwise' to the two eight bit values. The meaning of 'bitwise' is best explained by an example. If the A register contains $D5 and the memory location contains $E7 then the result of ANDing them together is -

A register	1	1	0	1	0	1	0	1
memory	1	1	1	0	1	1	1	0
result	1	1	0	0	0	1	0	0

or in other words, $C4. You can see that there is only a 1 in the result when both the A register and the memory location contain a 1 in the same place. This is because the result is obtained by applying the truth table to each pair of corresponding bits i.e to the first bit in A and the first bit in the memory location and so on.

Of the addressing modes that have been introduced so far, the ANDA and ANDB instructions can be used with immediate, direct and extended. So,

```
ANDA #$34
```

and

```
ANDB $4020
```

are both legal examples.

ORA address and ORB address

The ORA and ORB instructions work in a similar way to the AND

instruction, but they produce the bitwise OR of the current contents of the A or B register and a memory location and leave the result in the register. The only difference is that the truth table used is -

reg.	mem.	result
0	0	0
0	1	1
1	0	1
1	1	1

If you examine the table carefully you will see that the result is 1 if either of reg. or mem. are 1. As an example of the OR instruction suppose that the B register contains $E3 and the memory location contains $78. The result would be -

B register	1	1	1	0	0	0	1	1
memory	1	1	1	1	1	0	0	0
result	1	1	1	1	1	0	1	1

or $FB. Notice that there is a 1 in the result wherever either value has a 1. The ORA and ORB instructions support the immediate, direct and extended addressing modes, so

```
ORB #$F3
and
ORA $2950
```

are valid examples.

EORA address and EORB address

The 'exclusive OR' is not as well known as the OR operation but it is just as useful. The two instructions EORA and EORB perform the exclusive OR operation on the contents of the A or B register and a memory location, leaving the result in the register. The exclusive OR is taken bitwise according to the following truth table -

reg.	mem.	result
0	0	0
0	1	1
1	0	1
1	1	0

Notice that the result is a 1 only when one of reg. or mem. is 1. That is the result is 1 if either but not BOTH is a 1. As an example of the EOR instruction suppose the A register contains $CE and the memory location contains $5F then the result is -

A register	1	1	0	0	1	1	1	0
memory	0	1	0	1	1	1	1	1
result	1	0	0	1	0	0	0	1

or $91. Notice that there is a 1 in the result only where there is exactly one 1 in either of the register or memory values. You can use all the addressing modes that we have met so far (with the exception of inherent addressing) with EORA and EORB so

```
EORA #45
```

and

```
EORA $A32A
```

re both valid examples.

COMA,
COMB and
COM address

The COM (standing for COMplement) is the assembly language equivalent of the BASIC NOT. It is different from the previous three logical operations in that it works on a single value. COMA changes the current value in the A register so that 0s become 1s and 1s become 0s (as always, the result it left in the register). COMB carries out the same operation on the value stored in the B register. In both cases there is no need to specify an address in the address field because only the value in the A or the B register is involved in the operation. Like the RTS instruction both COMA and COMB use the inherent addressing mode. The truth table for the COM operation is particularly simple -

reg.	result
0	1
1	0

and this is applied bitwise as usual. For example, if the A register contains $6F then the result of COMA is

register	0	1	1	0	1	1	1	1
result	1	0	0	1	0	0	0	0

or $90. The only addressing mode that can be used with COMA and COMB is inherent and the BASIC assembler can already deal with this mode.

The COM operation also has an extra form - COM address. This instruction will perform the COM operation on the memory location given by 'address'. This seems to invalidate an earlier fundamental observation that for any operation to take place the data must be brought from memory into the CPU. For example, COM $7F00 will complement the contents of memory location $7F00 without the intervention of the A or the B register. In fact this 'direct' memory modification is nothing but an illusion because the contents of the memory are brought into memory and stored in a nameless temporary register, operated on and then restored back in the original memory location. (This nameless register is of no great interest from the programmer's point of view, in that you cannot use it in any other situation, but it is nevertheless there). This apparent direct modification of memory by the COM instruction is something that is available as an extra with nearly all of the 6809's instructions that operate only on a single data value. An example of the COM instruction is-

```
COM $7032
```

which complements the contents of memory location $7032. Of those we have met up to now, COM can be used with the direct and extended addressing modes.

Adding logic to the BASIC assembler

All that is needed to extend the BASIC assembler to handle the 6809's complement of logical operations is the addition of the appropriate DATA statements -

```
  1 REM BASIC ASSEMBLER V5.1

 19 DATA ANDA,&H84,&H94,&HA4,&HBB,-1
 20 DATA ANDB,&HC4,&HD4,&HE4,&HF4,-1
 21 DATA ORA,&H8A,&H9A,&HAA,&HBA,-1
 22 DATA ORB,&HCA,&HDA,&HEA,&HFA,-1
 23 DATA EORA,&H88,&H98,&HA8,&HB8,-1
 24 DATA EORB,&HC8,&HD8,&HE8,&HF8,-1
 25 DATA COMA,-1,-1,-1,-1,&H43
 26 DATA COMB,-1,-1,-1,-1,&H53
 27 DATA COM,-1,&H03,&H63,&H73,-1
```

Bit manipulation

The principal use of the logical operators is in changing individual bits within a memory location. Usually the problem is that a particular bit or group of bits has to be set to 0 or 1 without altering the rest. This is usually referred to as 'bit manipulation'.

Setting any bit to zero is easy once you recall that the result of an AND is only one if both of the corresponding bits in the register and memory are 1. Suppose the problem is to set b5 and b2 to zero in memory location $7F00 which currently contains $36 -

b7	b6	b5	b4	b3	b2	b1	b0
0	0	1	1	0	1	1	0

(Notice that the bits in a memory location are numbered starting from zero.) Then, ANDing the memory location with the bit pattern -

1 1 0 1 1 0 1 1

produces the result

0 0 0 1 0 0 1 0

which is the same as the original contents of the memory location except that b5 and b2 are 0. If you look at the value that the memory location was ANDed with then you should see that it is a sort of 'picture' of what we

wanted to happen, in the sense that at each bit position where nothing was to be changed there was a 1 and at each bit position to be zeroed there was a 0. For example, to zero b7,b5 and b1 you would AND the contents of the memory location with

b7	b6	b5	b4	b3	b2	b1	b0
0	1	0	1	1	1	0	1

or $5D. If the memory location in question was $7F00 then an assembly language program to achieve this would be

```
LDA$7F00
ANDA #$5D
STA$7F00
```

Notice the way that the A register is used to form the AND of the memory location and the immediate data.

The second problem of bit manipulation, setting a bit or a group of bits to 1 while leaving the rest unaltered, can be solved in roughly the same way using OR. In this case the contents of the memory location have to be ORed with a bit pattern that is zero apart from the bit positions that you want to set to 1. For example, if you want to set b4, b3 and b0 to 1 in memory location $7FF0 which currently contains $54 -

	b7	b6	b5	b4	b3	b2	b1	b0
$54=	0	1	0	1	0	1	0	0

then it should be ORed with

0 0 0 1 1 0 0 1

or $19 which gives -

0 1 0 1 1 1 0 1

or $5D. A program to carry this out would be -

```
LDA$7FF0
ORA #$19
```

```
STA $7FF0
```

Apart from setting a group of bits to 0 or 1 it is occasionally necessary to 'flip' a bit or a group of bits. In other words if a bit is currently 1 it should be changed to 0 and vice versa. This action can be achieved using the EOR operation. In this case the contents of the memory location have to EORed with a value that is made up of 0s apart from the bit positions that have to be 'flipped'. For example, to flip b6, b5 and b3 of memory location $7FF0 which currently contains $A5 or

b7	b6	b5	b4	b3	b2	b1	b0
1	0	1	0	0	1	0	1

it has to be EORed with

0 1 1 0 1 0 0 0

or $68 to give the result -

1 1 0 0 1 1 0 1

or $CD. The assembly language program to carry this out would be-

```
LDA$7FF0
EORA #$68
STA$7FF0
```

You should be able to see that there is a pattern in these three ways of manipulating bits. Each time the contents of a memory location are combined using AND, OR or EOR with an immediate data with a particular bit pattern. The immediate data is usually called a 'mask' and the rules for construction and using a mask can be summarised as follows:

1) To set a group of bits in a data value to 0 without affecting the rest, construct a mask with all 1s apart from the bit positions that have to be set to 0, then AND it with the original value.

2) To set a group of bits in a data value to 1 without affecting the rest, construct a mask with all 0s apart from the bit positions that have to be set to 1, and OR it with the original value.

3) To flip a group of bits in a data value without affecting the rest, construct a mask with all 0s apart from the bit positions that have to be flipped and EOR it with the original value.

Of course if you want to flip all of the bits in a memory location or in either of the A or B registers then you can dispense with masks and just use the COM instructions. All bit manipulation that you might want to do can be carried out using combinations of these operations.

The shift instructions

The logical operations change bits at particular positions within the bit pattern. The shift instructions are also used in bit manipulation but rather than changing bits at given bit positions they move the whole bit pattern to the right or to the left - hence the name 'shift operations'. To understand the way that shifts work it is important that you realise that a memory location and the A and B registers always hold exactly eight bits, no more and no less. As an example of a shift operation, consider a simple shift of all the bits one place to the left. In other words, b7 takes on the value of b6, b6 takes the value of b5 and so on. This is perfectly simple but there are two questions that have to be answered. What happens to the old value of b7 and what value does b0 take on? If the bit pattern was $C3 i.e. -

b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	0	0	0	1	1

then after shifting one place to the left we have

		b7	b6	b5	b4	b3	b2	b1	b0	
1	<-	1	0	0	0	0	1	1	?	<-

As you can see the old value of b7 has 'fallen' off the end of the number and there is a problem about what value should be 'shifted into' b0. There are only two ways that shift instructions can vary -

in, their direction i.e. right shifts and left shifts

and, in the way that they solve the problem of what happens to b7 and what value should be shifted into b0

It is worth keeping these two things in mind when reading the following descriptions of the 6809's range of shift operations.

The logical shift instructions, LSL and LSR

The characteristic that marks out the logical shift instructions is that the bit that is shifted 'off the end' of the number is stored in a special register inside the CPU and a 0 is shifted into the 'other end'. This special register is called the condition code register and is fully described in the next chapter. All that you need to know to understand how shift instructions work is that a single bit of this register - the C or 'Carry bit' - is used to hold the bit that would otherwise be lost as a result of the shift.

There are three forms of the LSL (Logical Shift Left) instruction-

LSLA
LSLB

and

LSL address

The first two perform the LSL operation on the A and B register respectively and the third on the memory location at 'address'. The LSL operation can be imagined as -

C ← b7 b6 b5 b4 b3 b2 b 1 b0 ← 0

indicating that all the bits are moved one place to the left, that b7 is shifted into the C bit and a 0 is shifted into b0. For example, if the A register contains $C5 or -

b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	0	0	1	0	1

then the result of LSLA is -

C	b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	0	0	1	0	1	0

or $8A with a 1 stored in the C bit. Notice that LSLA and LSLB are both inherent but the direct and extended addressing modes can be used with LSL address.

There are also three forms of the LSR (Logical Shift Right) instruction -

```
LSRA
LSRB
```

and

```
LSR address
```

The operation of the LSR instruction is roughly the same as the LSL instruction except of course that all the bits are moved to the right. In this case it is b0 that is stored in the C bit and the 0 is shifted into b7. This is best imagined as -

```
0--> b7 b6 b5 b4 b3 b2 b1 b0 --> C
```

For example, if the A register contains $C5 or

b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	0	0	1	0	1

then the result of LSRA is -

b7	b6	b5	b4	b3	b2	b1	b0	C
0	1	1	0	0	0	1	0	1

or $62 with 1 stored in the C bit. All the other details of the LSR instructions are the same as for the LSL instructions.

2 The rotate instructions, ROR and ROL

The rotate instructions also make use of the C bit to store the bit that 'falls off' the end but they also use it as the source of the bit shifted in at the other end.

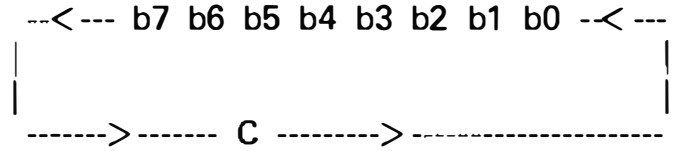
There are three forms of the ROL (ROtate Left) instruction -

```
ROLA
ROLB
```

and

```
ROL address
```

which ROL the A register, the B register and the contents of the memory location at 'address'. The ROL operation is best imagined as -



in other words each bit moves one place to the left as in a LSL but the value of the C bit is moved into b0 and b7 is moved into C. If you look at the diagram you will see why the term rotate is used in ROL. For example, if the A register contains $C5 -

b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	0	0	1	0	1

and the C bit is 0 then after a ROLA the result is -

```
1 0 0 0 1 0 1 0
```

or $8A and the new value of the C bit is 1.

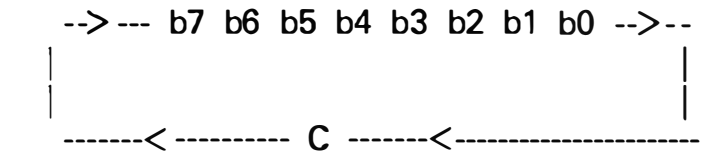
There are also three forms of the ROR (ROtate Right) instruction-

```
RORA
RORB
```

and

```
ROR address
```

which apply the ROR operation to the A register, the B register and the memory location at 'address' respectively. The ROR instruction works in the same way as ROL only the bits are moved to the right. The best way to think of the ROR operation is -



In other words each bit moves one place to the right, the value of the C bit is moved into b7 and b0 is moved into the C bit. For example, if the A register contains $C5 or

b7	b6	b5	b4	b3	b2	b1	b0
1	1	0	0	0	1	0	1

and the C bit is 1 the result of RORA is -

1 1 1 0 0 0 1 0

or $E2 and the new value of the C bit is 1.

The key feature of the ROL and ROR instructions is that they both involve the current value of the C bit. So far the only way that we have of determining the C bit is via the use of the shift instructions themselves. Instructions to change the C bit directly are discussed in the next chapter and these make the ROR and ROL instructions much more useful.

The 6809 does have another shift instruction but this is also better described as part of the next chapter, on arithmetic.

Adding shifts to the BASIC assembler

Once again the shift instructions present no problems for the BASIC assembler and adding them is simply a matter of including the new DATA statements.

```
 1 REM BASIC ASSEMBLER V5.2

28 DATA LSLA,-1,-1,-1,-1,&H48
29 DATA LSLB,-1,-1,-1,-1,&H58
```

```
30 DATA LSL,-1,&H08,&H68,&H78,-1
31 DATA LSRA,-1,-1,-1,-1,&H44
32 DATA LSRB,-1,-1,-1,-1,&H54
33 DATA LSR,-1,&H04,&H64,&H74,-1
34 DATA ROLA,-1,-1,-1,-1,&H49
35 DATA ROLB,-1,-1,-1,-1,&H59
36 DATA ROL,-1,&H09,&H69,&H79,-1
37 DATA RORA,-1,-1,-1,-1,&H46
38 DATA RORB,-1,-1,-1,-1,&H56
39 DATA ROR,-1,&H06,&H66,&H76,-1
```

Labels and data - RMB, FCB and FDB

In the last chapter the idea of using a label to represent an address was introduced. It may not have been clear from the description that an address label can be used anywhere that an address can. In particular, as well as being used in JMP instructions to specify the 'destination' address labels can be used to specify the location of data in operations such as AND etc. For example, suppose you are using memory location $7FF0 to store some data, then to load it into the A register you could use-

```
LDA $7FF0
```

but perhaps -

```
@DATAEQU$7FF0
LDA@DATA
```

is easier to read and understand. Also, if you suddenly decide on a major program change that moves the location that the data is stored to $7000 then the only change that you have to make to the second version is -

```
@DATA EQU $7000
```

In short, labels are just as useful when used to represent the addresses of data as destination addresses in JMP instructions. However, it is very important to preserve the distinction between immediate addressing and extended addressing. For example

```
LDA #@LABEL
```

will load the A register with the value of @LABEL and

```
LDA @LABEL
```

will load the A register with the value stored in the memory location whose address is the value of @LABEL.

This idea of using labels for data is worth extending by the use of three new pseudo ops - RMB (Reserve Memory Bytes), FCB (Form Constant Byte) and FDB (Form Double Byte). When a label is used as a destination address it isn't defined using the pseudo op EQU, instead it is defined by the position that it occupies in the program - i.e. by the instruction that it labels. This idea can be used to define labels that 'mark' memory locations used to store data. For example, suppose you want to use three memory locations to store some information and you don't really mind exactly where they are as long as they are out of the way of the program and you know exactly where to find them. The RMB pseudo op can be used to reserve any number of memory locations anywhere in a program. For example,

```
LDA@DATA
ANDA #$0F
JMP@LOOP
@DATARMB3
```

the RMB instruction reserves 3 memory locations for data storage at the end of the program and the label @DATA is equal to the address of the first of them. The effect of-

```
label RMB n
```

is simply to reserve the next n memory locations for data and set the value of the label to the address of the first one. In many ways RMB looks just like a standard machine code instruction that takes n memory locations to store!

RMB is useful for reserving memory (for large tables, for example) but often in a program it would be an advantage to define a single memory location as holding data and also initialise it to some value. This is exactly what the FCB pseudo-op does -

```
@DATA1 FCB $76
```

FCB stores $76 in the next free memory location and sets the label @DATA1 to its address. It is important to be clear that FCB is not an instruction that the 6809 carries out - after all it is a pseudo-op, an instruction to the assembler to do something. What happens when the assembler meets

label FCB value

is that it stores 'value' in the next memory location (i.e. the one that it would have used to store the machine code of the next instruction) and sets 'label' to its address. When the assembled machine code program is run, the only effect that the FCB has had is that memory location 'label' contains 'value'. The pseudo-op FDB works in more or less the same way as FCB except that it uses two memory locations to store its value and the label is set to the address of the first of the two memory locations.

Adding these pseudo-ops to the BASIC assembler is easy, if a little long winded. The existing test for the pseudo-op EQU at line 4001 has to have tests for RMB, FCB and FDB added to it and subroutine 6500 (the pseudo-op handler) has to be extended.

```
   1 REM BASIC ASSEMBLER V5.3

4002 IF M$="RMB" THEN PS=2:RETURN
4003 IF M$="FCB" THEN PS=3:RETURN
4004 IF M$="FDB" THEN PS=4:RETURN


6500 IF PS<>1 THEN GOTO 6540

6540 IF PS<>2 THEN GOTO 6570
6545 IF PASS=2 THEN PRINT TAB(15);A$(I)
6550 P=P+A
6560 IF PASS=2 THEN GOTO 6520

6570 IF PS<>3 THEN GOTO 6650
6580 A=A-INT(A/256)*256
6590 IF PASS=1 THEN GOTO 6620
6600 PRINT HEX$(P);TAB(5);HEX$(A);
6610 PRINT TAB(15);A$(I)
```

```
6620 POKE P,A
6630 P=P+1
6640 RETURN

6650 IF PS<>4 THEN RETURN
6660 IF PASS=1 THEN GOTO 6710
6670 LB=A-INT(A/256)*256
6680 HB=INT(A/256)
6690 PRINT HEX$(P);TAB(5);HEX$(HB);TAB(8);HEX$(LB);
6700 PRINT TAB(15);A$(I)
6705 POKE P,HB:POKE P+1,LB
6710 P=P+2
6720 RETURN
```

Lines 4002-4004 detect the new pseudo-ops and set the variable PS to a value that indicates which one has been found. Lines 6540-6560 process RMB, the only effect of which is to increase the value of P by the number of memory locations to be reserved. Notice that any label used with RMB (or FCB and FDB) will be processed in the normal way, just as if the new pseudo-ops were standard mnemonics and the 'value' that follows the pseudo-ops will be processed as an address field and stored in the variable A. Lines 6570-6640 process an FCB pseudo-op. The only point to notice here is that the value POKEd into memory is truncated by line 6580 to be in the range 0 - 255. In other words, an fcb like FCB $1234 will be interpreted by the assembler as FCB $34. Lines 6650-6720 process a PDB pseudo-op. This is done in much the same way as for the FCB, only now two values are POKEd into memory HB, and LB representing the 'high' or 'most significant' byte and the 'low' or 'least significant' byte of the value respectively. Quite a few of the statements in the pseudo-op handler look complicated but are simply formatting and printing the results of the assembly so that you can keep track of what is going on.

If you are a little unsure of the way that RMB, FCB and FDB are used then you will find plenty of examples in forthcoming chapters!

Summary

1) The logical operators AND, OR, EOR and COM have been described in this chapter along with the logical shifts LSL, LSR, ROR

2) The idea that the 'bit pattern' is the fundamental type of data that assembly language programs work with has been emphasised by a discussion of bit manipulation using the logical operators.

3) Finally the use of labels to mark the location of data was introduced along with the pseudo-ops RMB, FCB and FDB.

Micro Projects

1) Some information is stored in an memory location labelled by @DATA. Write a short program that will set b6 to 1 and b3, b2 and b1 to 0 without affecting any other bits.

2) The value $0F is stored in a memory location labelled by @FLIPPER. Write a program in the form of a loop that will change the value stored in @FLIPPER to $F0 the first time through and then to $0F the second time through, and so on alternating the value $0F with $F0.

3) How could you use LSL and ROL to move the top four bits in the A register into the bottom four bits in the B register? (In other words b7, b6, b5 and b4 of the A register should occupy b3, b2,b1 and b0 respectively of the B register.)

4) Write a program which defines two memory locations as data using the FCB pseudo-op and then ANDs their contents together placing the result in yet another memory location.

Chapter Six

Arithmetic Instructions

This chapter examines the ways in which the 6809 can handle bit patterns that represent numbers. It is important that you are clear about the way that numbers can be represented by bit patterns and what sorts of things can go wrong with seemingly simple arithmetic. Fortunately there is no need to understand and actually be able to do binary arithmetic because the 6809 can do it for you! As you read this chapter, try to keep in the back of your mind that the fundamental data for assembly language programming is the bit pattern and everything else, including numbers, is a result of the way that you are using bit patterns.

Assembly language arithmetic

The use of the ADDA and ADDB instructions to add two numbers together has already been described in earlier chapters. However, there is more to 6809 assembly language arithmetic than these two instructions. In particular, there is the question of working with numbers larger than $FF and that of working with negative numbers. Before these and other aspects of arithmetic are explained it is worth explaining the role that arithmetic plays in assembly language programming. It is probably true to say that in BASIC, arithmetic is basic! But this is not so in assembler. It takes many assembly language instructions to perform the apparently simple sum

2.321/1.422

that BASIC would accept in one go or even as part of another instruction! The result of the high level of difficulty encountered with assembly language arithmetic is that complex calculations are rarely found in assembly language programs. If you are planning to write a program that needs much in the way of calculation, then it is better to think in terms of BASIC or a mixture of

assembly language and BASIC. If you really do need the undoubted speed advantage of assembler in doing complicated calculations than you will need to use a very special collection of assembly language subroutines called a 'software floating point package'. In most cases, however, the sort of application that assembly language is put to only needs the addition, subtraction and occasionally multiplication of fairly small numbers and this is not so difficult.

Negative binary numbers - two's complement.

So far the only sort of numbers that we have been able to deal with are positive numbers in the range 0 to $FF. There is more than one way to represent a negative number in binary but the most common and the one used by the operations in the 6809 is the so-called 'two's complement form'. There is an extensive theory behind two's complement negative numbers but all that you need to know to use the negative numbers in assembly language is their format.

If you ask yourself what makes a number like -6 a negative number, the answer is that if you add it to +6 the answer is 0. That is to say, the fundamental property that makes a number negative is that, if x is a number then its negative, i.e. -x, is a number that when added to x gives the answer zero. This definition of a negative number seems to be interesting but hardly useful until you recall an observation made in Chapter Four. If you add one to a memory location or a register you will eventually reach the maximum value that can be stored, i.e. $FF. The question is what happens if you add one to this maximum value? The answer is not that you get an error message, instead the result is zero.

In other words, 6809 arithmetic works on eight-bit numbers in such a way that if you start at zero and repeatedly add one you will reach $FF and then 'roll over' to zero and carry on round again, just like the mileometer on a car or a bicycle. The importance of this observation is you can add two binary numbers together and get the result zero. For example, 2 + 254 (in decimal) gives the result0 as2 + 253 gives the result255 and adding 1 to this makes the value roll over to0. In this special sense 254 behaves as if it was the negative of 2. For any number between 0 and 127 it is possible to find another number greater than 127 that produces a sum of zero.

Now you should be able to see how negative numbers can be represented in binary in such a way that they play their correct part in arithmetic. Instead of using the whole range 0 to 255 as representing positive numbers the range is divided into two halves, the positive numbers from 0 to 127 and the 'negative' numbers from 128 to 256. This representation of negative numbers as the second half of the total range is known as 'two's complement representation'. Its single great advantage is that if you use it you can carry out any arithmetic that you like without worrying about whether a number is positive or negative - that aspect looks after itself! Notice that this isn't true of the usual decimal way of handling negative numbers. For example, if you do the sum 3 + 4, then you add the two numbers; but the sum 3 + (-4) would be done by subtraction. In binary, both sums would be done using addition, the first 3 +4 in the usual way but 3 + (-4) would be done by finding the two's complement of 4 and then ADDING it to 3.

You needn't worry about the details of carrying out arithmetic in two's complement form because the 6809 will look after you, but you do need to know the essential features of a two's complement number. Any number between 0 and $7F (0 - 127 in decimal) is a regarded as a positive number. Putting this another way, any binary number that has b7 equal to 0 is regarded asa positive number. Any number between $80 and $FF (128 -255 in decimal) is regarded as a negative number. Putting this another way, any binary number that has b7 equal to 1 is regarded as a negative number. Thus, if you are using two's complement arithmetic all you have to do to decide if a number is positive or negative is to look at b7 - if it's 0 then the number is positive, if it's 1 then the number is negative. For example,

b7	b6	b5	b4	b3	b2	b1	b0
0	1	0	1	1	0	0	0

is a positive number equal to $58 and

b7	b6	b5	b4	b3	b2	b1	b0
1	0	1	1	0	1	0	1

is a negative number numerically equal to $B5, or 185 in decimal, and (as 71 + 185 =0 using eight bit arithmetic that rolls over at 255) it represents-71. Don't worry about converting negative numbers to their standard decimal representation and vice versa, because the 6809 contains instructions that will do it for you.

The representation of negative numbers in two's complement form is not a difficult idea. However, it does contain an illustration of the subtle point about bit patterns being the fundamental data for assembly language. The example of a two's complement number given above is the representation of -71, however, it is also the binary representation of 185. You should be able to see that the question "which is correct?" is not sensible. Which of the two numbers the bit pattern 10110101 represents depends on how you view it - as a two's complement number or a simple binary number. This is an important point because, as far a 6809 arithmetic is concerned, two's complement numbers and simple binary numbers are treated in the same way. The only difference arises when you come to interpret the answer. If, at the start of the arithmetic, you considered the numbers to be simple binary then you must interpret the answer as simple binary, but if you regarded the numbers as two's complement then the answer must be interpreted as two's complement - it's all up to you. So, if you are going to need negative numbers, then select the two's complement form. Otherwise stick to simply binary arithmetic.

The ADD and SUB instructions - the D register

The instruction ADD has already been introduced in earlier chapters. However, as well as its ADDA and ADDB forms it can also be used to add numbers in the range 0 to $FFFF. To make this possible the A and B register have to be used together as a single register capable of holding 16 bits. This giant register is called the D register but this new name shouldn't be allowed to obscure the fact that the D register is nothing more than the familiar A and B registers working together. The instruction -

```
ADDD address
```

adds the contents of the memory locations at 'address' and at 'address + 1' to the current contents of the D register, leaving the result in the D register. Notice that two memory locations are used to store the value that is added to the D register and, in line with the way that extended addresses are stored, the most significant eight bits are stored in 'address' and the least significant eight in 'address'+ 1. In the case of the D register, the A register holds the eight most significant bits and the B register holds the eight least significant bits. These two facts can be illustrated thus -

```
data  = address | address + 1
D     =       A | B
```

Apart from the advantage of working on 16-bit numbers, the ADDD instruction is very similar to the ADDA and ADDB instructions. It does, however, pose a slight problem for the BASIC assembler in that -

```
ADDD #$FE72
```

looks like a standard immediate mode instruction but, as the D register is 16 bits long, the immediate data is also 16 bits or 2 memory locations. This means that the assembler now has to detect such 16-bit operations and POKE the correct amount of DATA. There are other instructions that use the D register and there are other 16-bit registers in the 6809 so this modification to the assembler will be left until later. As well as the ADDD instruction there are also the LDD and STD instruction which can be used to load and store the register. As well as the three ADD instructions, ADDA, ADDB and ADDD the 6809 also has a corresponding set of three SUB (SUBtract) instructions - SUBA, SUBB and SUBD. These instructions can be used with the same addressing modes that can be used with the ADD instruction and work in roughly the same way, except of course that they subtract the contents of memory from the current contents of the register, leaving the result in the register.

Arithmetic with simple binary numbers

The ADD and SUB instructions can be used to carry out arithmetic on simple binary numbers as long as the result is in the same range as the original numbers. For example,

```
LDA #$0A
ADDA #$06
```

will correctly add $0A to $06 leaving the result $10 in the A register or -

```
LDA #$0A
SUBA #$06
```

which will correctly subtract $06 from $0A leaving the result $04 in the A register. You can carry out any arithmetic with the A register as long as the numbers are in the range 0 to $FF and get a correct answer as long as the result is also in this range. If the result is outside this range then the 'roll over'

behaviour will make a nonsense of the answer. This is called 'arithmetic overflow'. For example,

```
LDA #$F0
ADDA #$15
```

leaves the result 5 in the A register which is certainly not correct and

```
LDA #$06
SUBA #$07
```

leaves the result $FF in the A register which once again is far from correct. When performing arithmetic on bit patterns that you are considering as simple binary numbers, the rule is that the result must be in the same range as the original numbers. For arithmetic on the A and B registers the range is 0 to $FF and for the D register the range is 0 to $FFFF. Notice in particular this rules out any negative results for, after all, negative numbers are no part of simple binary!

You may be worried about the possibility of carrying out some arithmetic in a program and using invalid results because of overflow. This problem will be dealt with in the next chapter where instructions that can be used to test for overflow will be described.

Arithmetic with two's complement numbers - the NEG instruction

The rule for carrying out arithmetic with two's complement numbers is the same as for simple binary numbers in that the result must lie in the same range as the numbers involved in the arithmetic. The main difference is that, of course, now we can work with negative numbers and get negative results. For example,

```
LDA #$06
SUBA #$07
```

gives the result $FF which is the two's complement form of -1, which is correct. Notice that the result is only correct because we are working with two's complement numbers. You might be thinking that this distinction between two's complement and simple binary is something that can be

ignored until you get a negative result. However, if you are using two's complement representation

```
LDA #$FF
ADDA #$05
```

has to interpreted as adding $FF or -1 to $05 which when you take the roll over into count give the answer $04 which is quite correct. However, if you interpret the numbers as simple binary then the answer is the same, i.e. $05, but it isn't the correct answer to $FF+ $05! Once again the point is that the arithmetic is the same no matter what the bit patterns represent but the meaning of the answer depends on whether you are using simple binary or two's complement. You can still get an overflow while using two's complement if the result is outside the range of numbers that can be represented. How to detect a two's complement overflow will be described in the next chapter.

You may be wondering how to convert a negative number to its two's complement representation. This is quite easy to achieve using the NEG instruction. There are three forms of the NEG instruction -

```
NEGA
NEGB
```

and

```
NEG 'address'
```

which will perform the NEG operation on the A register, the B register and the contents of the memory location at 'address' respectively. (Notice that there is NO NEGD instruction.) The NEG operation changes a two's complement positive number into a two's complement negative number. For example,

```
LDA #$01
NEGA
```

results in $FF in the A register which is of course the two's complement representation of - 1. One of the most useful facts about the NEG operation is that it will not only change a positive number into a negative number but vice versa. This is rather like the familiar -(-3) giving the result 3. For example,

```
LDA #$FF
NEGA
```

results in $01 in the A register. So, using NEG you can easily convert between positive and negative numbers.

To summarise -

1) Using two's complement representation you can do arithmetic with positive and negative numbers.

2) The valid range for eight-bit two's complement numbers is -128 to +127 and for 16-bit two's complement numbers is -32768 to +32767.

3) The result of a two's complement operation is only valid if the correct result lies in the range that can be represented.

4) The NEG instruction can be used to convert between positive and negative eightbit two's complement numbers.

The CLR, INC and DEC instructions

The three instructions CLR, INC and DEC are in some senses unnecessary in that they perform operations that can be carried out using other instructions. However, they are very convenient to use and, as will be explained in the next chapter, they do have a slightly different action than their equivalents.

The CLR (CLeaR) instruction has three forms -

```
CLRA
CLRB
```

and

```
CLR 'address'
```

which perform the CLR operation on the A register, the B register or the contents of the memory location at 'address' respectively. The CLR operation is simplicity itself, it loads the register or the memory location with 0 (i.e. eight zeros).

The INC (INCrement) instruction has three forms -

```
INCA
INCB
```

and

```
INC 'address'
```

which perform the INC operation on the A register, the B register and the memory location at 'address' respectively. The INC operation simply adds one to the register or the memory location.

The DEC (DECrement) instruction also has three forms -

```
DECA
DECB
```

and

```
DEC 'address'
```

which performs the DEC operation on the A register, the B register and the memory location at 'address' respectively. The DEC operation, as you might suspect, subtracts one from the contents of the register or memory location.

These three operations CLR, INC and DEC are faster and take less memory than their equivalents and are nearly always to be preferred.

Adding arithmetic to the BASIC assembler.

Adding the instructions that have been introduced so far is mainly a matter of including the appropriate DATA statements. The one exception is the introduction of the operations on the D register that allow immediate addressing i.e. LDD,ADDD and SUBD. Currently the BASIC assembler can only handle eight-bit immediate addressing. All that has to be done to cope with the possibility of 16-bit immediate data is to test for the presence of "D" at the end of an instruction and then POKE two memory locations, the first

with the most significant byte and the second with the least significant byte of the value of the address field.

```
  1 REM BASIC ASSEMBLER V6.1

 40 DATA ADDD,&HC3,&HD3,&HE3,&HF3,-1
 41 DATA SUBA,&H80,&H90,&HA0,&HB0,-1
 42 DATA SUBB,&HC0,&HD0,&HE0,&HF0,-1
 43 DATA SUBD,&H83,&H93,&HA0,&HB0,-1
 44  ATA CLRA,-1,-1,-1,-1,&H4F
 45 DATA CLRB,-1,-1,-1,-1,&H5F
 46 DATA CLR,-1,&H0F,&H6F,&H7F,-1
 47 DATA INCA,-1,-1,-1,-1,&H4C
 48 DATA INCB,-1,-1,-1,-1,&H5C
 49 DATA INC,-1,&H0C,&H6C,&H7C,-1
 50 DATA DECA,-1,-1,-1,-1,&H4A
 51 DATA DECB,-1,-1,-1,-1,&H5A
 52 DATA DEC,-1,&H0A,&H6A,&H7A,-1
 53 DATA NEGA,-1,-1,-1,-1,&H40
 54 DATA NEGB,-1,-1,-1,-1,&H50
 55 DATA NEG,-1,&H00,&H60,&H70,-1
 56 DATA STD,-1,&HDD,&HED,&HFD,-1
 57 DATA LDD,-1,&HCC,&HDC,&HEC,&HFC,-1

6024 IF TYPE=1 AND RIGHT$(M$,1)="D" THEN TYPE=4
```

The only line that needs any explanation is 6024 which checks for a "D" at the end of the mnemonic in immediate mode i.e. when TYPE=1. If this is detected, TYPE is changed to 4 so that two memory locations will be POKEd with the value in A (see line 6040 in the existing program).

Extended precision arithmetic

So far, the only arithmetic that has been described in detail is eight-bit simple or eight-bit two's complement arithmetic using the A and B registers. As already briefly discussed, you can use the A and B registers together - as the D register - but the range of operations on the D register is rather limited. However, the D register is the simplest way of carrying out 16-bit arithmetic. If you use the 16 bits to represent only positive numbers i.e. simple binary

then you can cope with a range of 0 to 65535. If you use 16-bit two's complement then the range is -32768 to +32767 In either case the range is often large enough not to have to worry about extending it any further. For example, if you want to add two numbers that are too large to be represented in eight bits but are within the 16-bit range then -

```
LDD @NUM1
ADDD @NUM2
STD @NUM3
```

where @NUM1, @NUM2 and @NUM3 each label the first of TWO memory locations that hold the 16-bit numbers. Subtraction can be achieved by replacing the ADDD @NUM2 instruction by SUBD @NUM2 instruction. All the comments about overflow and two's complement form apply to arithmetic on the D register if you take into account the much increased range. So for example, $FFFF is the 16-bittwo's complement representation of -1 (because adding 1 to $FFFF makes it 'roll over' to 0).

The lack of a NEGD instruction makes it a little difficult to obtain the two's complement form of a number. This little problem is easy to solve once you recall that 0-x is -x. So to obtain the two's complement of a 16-bit number use-

```
LDD #0
SUBD@NUM
```

where @NUM labels the first of two memory locations that holds the value.

Another problem that often occurs is combining eight-bit and 16 results. For example, if during a program you have been calculating something that can easily be represented correctly by eightbits, then it is a waste of time and memory to use 16 bits when 8 will do. However, if at the end of the calculation the result has to be added to or subtracted from a 16-bit number how can this be done? The solution is simply to load the eight-bit value into the B register ,clear the A register and carry out the 16-bit arithmetic using the D register. For example,

```
LDB@SMALL
CLRA
ADDD @LARGE
STD @ANS
```

where @SMALL labels a single memory location holding the eight-bit value, @LARGE labels the first of two memory locations holding the 16-bit value and @ANS labels the first of two memory locations used to store the 16-bit result. You could subtract the two numbers simply by changing the ADDD @LARGE to SUBD @LARGE. This method works for simple binary numbers because, to change an eight-bit value into, a 16-bit value all you have to do is write eight zeros in front. However, things are not so simple for negative two's complement numbers. For example, $FF is eight-bit two's complement for -1 but adding eight zeros in front gives $00FF which is 16-bit two's complement for 255. The correct thing to do is to add eight ONES to the front of a negative two's complement number. For example, writing eight ones in front of $FF gives $FFFF which is the 16-bit two's complement representation of -1. Obviously, for positive two's complement numbers the correct thing to do is to write eight zeros in front of the number and so it looks as though extending two's complement eight-bit to 16 bits is difficult. Once again the 6809 comes to our rescue - this time with the SEX (Sign EXtend) instruction. If you load the B register with an eight-bit two's complement number and than use the SEX instruction the A register will be loaded with eight zeros if the number in B is positive and eight ones if the number is negative, correctly extending the number. So, the two's complement equivalent of the previous eight-bit/16-bit addition is -

```
LDB @SMALL
SEX
ADDD@LARGE
STD@ANS
```

where the labels have the same meaning as before. Notice that if the values to be added were simple binary then using SEX to set the A register would give you an incorrect answer - so take care and think about which representation you are using.

Going back the other way, that is from 16 bits to eight bits, is also easy. If you are using simple binary then obviously if the result of an operation leaves the A register with nothing in it then you can ignore it and use the contents of the B register as a correct eight-bit result. In other words, taking eight zero bits off the front of a 16-bit number converts it into an eight-bit number. For example, $0032 is the same as $32 but $0132 is not the same as $32. For two's

complement representation things are just a little more complicated. If you throw away eight zero bits in front of a positive 16-bit two's complement number then you can get the wrong answer. For example, $00FF is 256 in 16-bit two's complement form but $FF is -1 in eight-bit two's complement form. The solution is that you have to reverse what the SEX instruction does in extending eight bits to 16. If the 16-bit number is positive and you can remove eight zero bits from the front and leave a positive eight-bit number, or if the 16-bit number is negative and you can remove eight ones from the front and leave a negative eight-bit number, then you will have the correct result. If you cannot do either of these then the 16-bit number is outside the range of representation by an eight-bit number. For example, $0004 is positive and removing eight zero bits leaves $04 which is still positive and both numbers represent 4, $FFFF is negative and removing eight ones leaves $FF which is still negative and both numbers represent -1. Fortunately it is not often that a 16-bit two's complement value has to be reduced to eight bits but this difficulty should emphasise how carefully you have to treat binary numbers.

Multiplication - the arithmetic shifts and MUL

Today's microprocessors are reasonably good at addition and subtraction but most of them make a very poor job of the other two operations, multiplication and division. The 6809, however, does at least have a simple multiplication instruction but even with this advanced microprocessor it is necessary to use a collection of special purpose subroutines if you want to do very much multiplication or division. Before dealing with the multiplication instruction it is worth introducing the easiest way of multiplying and dividing by two - the arithmetic shifts.

The logical shifts were introduced in the last chapter and the arithmetic shifts work in a similar way but their purpose is very different. If you are using a bit pattern to represent a simple binary number then shifting the pattern one place to the left is the same as multiplying the number by 2 and shifting it one place to the right is the same as dividing it by 2 and ignoring any fractional part. (Compare this to what happens to a decimal number when you multiply it by 10 or divide it by 10.) This means that LSL and LSR can be used to multiply and divide by 2. For example, if the A register contains $0A (10 in decimal) or

b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	1	0	1	0

then the result of a LSLA is

0 0 0 1 0 1 0 0

or $14 (20 in decimal) and the result of a LSRA (on the original $0A) is

0 0 0 0 0 1 0 1

or $05.

This use of the logical shifts is fine for simple binary numbers but when it comes to two's complement representation, things start going wrong. If you shift a negative two's complement number to the right using a LSR then a 0 is shifted into b7, which changes the number from negative to positive and this is not what is supposed to happen when dividing by 2. For example, if the number in the A register is $86 (which represents -122),

b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	0	0	1	1	0

then following a LSRA it contains

0 1 0 0 0 0 1 1

or $43 (which represents +67). The correct answer to -122 divided by 2 is -61 or in two's complement -

1 1 0 0 0 0 1 1

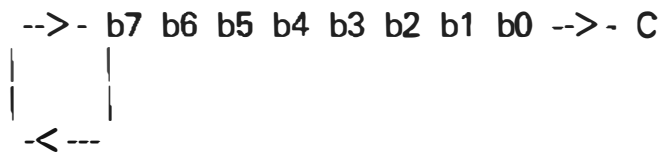
If you compare this number against the original you will see that to get the correct answer only needs a 1 to be shifted into b7 when the right shift is performed. However, this is not the whole answer because always shifting a 1 into b7 would give the wrong result when the number was positive. The correct solution is to perform the right shift in such a way that b7 keeps its value. This is the essence of an ASR (Arithmetic Shift Right) operation. The ASR operation is available in the usual three forms -

```
ASRA
ASRB
```

and

```
ASR 'address'
```

which perform the ASR operation on the A register, the B register and the contents of the memory location at 'address' respectively. The ASR operation is can be thought of as -



in other words, all the bits move one place to the right, b0 is stored in the C bit and b7 retains its current value.

Fortunately, there are no complications with the application of LSL to two's complement numbers but it is usual to allow the mnemonic ASL to be used to mean the same thing as LSL to make neat pairs of arithmetic shifts. In other words, shifting a two's complement number to the left one place and shifting a 0 into b0 (as in LSL) is the same as multiplying the number by 2. However, if multiplying the number by 2 would take it outside the range that can be represented then, just as with addition and subtraction, you will get an overflow (i.e. the wrong answer). For example, if the A register contains $20 or

```
b7   b6   b5   b4   b3   b2   b1   b0
 0    0    1    0    0    0    0    0
```

then following a ASLA instruction the A register contains -

```
0 1 0 0 0 0 0 0
```

or $40 which is twice $20. However, following a second ASLA the A register contains -

```
1 0 0 0 0 0 0 0
```

or $80 which is the two's complement representation for -128 which is definitely not twice $40 (+ 64 in decimal).

You may be wondering what all the fuss about multiplying and dividing by two is. The answer is that you can often multiply by small constants using nothing but shifts and adds. For example, to multiply the contents of a memory location by 5 you could multiply by 2 twice and then add the original number to the result i.e. if a is the number $5a = 4a + a$. Converting this to a program (assuming that the memory location to be multiplied is at $7F00) gives -

```
LDA$7F00
ASLA
ASLA
ADDA$7F00
STA$7F00
```

This program will multiply the contents of $7F00 but it is hardly complete in the sense that it makes no checks for an invalid result due to overflow: to do this would need instructions introduced in the next chapter!

For the general multiplication of two simple binary numbers the 6809 provides the MUL instruction. The MUL instruction multiplies the two simple binary numbers in the A and B registers and leaves the result in the D register. As the largest result that this can produce is $FFFF there is no chance of any sort of overflow and there is nothing that can go wrong when using the MUL instruction. For example, if the A register contains $0A (10 in decimal) and the B register contains $52 (82 in decimal) then following a MUL instruction the D register contains $0334 (820 in decimal). Notice that the MUL instruction will only give the correct answer if the two numbers to be multiplied together are in simple binary. In other words you can only directly multiply together positive numbers using MUL.

Binary Coded decimal - the DAA instruction

This section introduces a third method of representing numbers using bit patterns, 'Binary Coded Decimal' or 'BCD'. This representation is useful for doing small amounts of arithmetic on numbers that have been typed in from a keyboard and in situations where it is important that the 6809 does its arithmetic, for reasons of accuracy, in the same way that it would be done with pencil and paper,

The basic idea behind the BCD representation of a number is that using four bits you can represent the decimal digits 0 to 9 (using simple binary). Using this idea a number such as 2389 can be written as

```
  2    3    8    9
0010 0011 1000 1001
```

In other words each digit of the number is coded separately in binary as a group of four bits. This means that a single eight-bit memory location can be used to store a BCD number in the range 0 to 99 which should be compared to the range of a simple eight-bit binary number i.e. 0 to 256. P It should be obvious that you cannot add two BCD numbers and get the correct answer in BCD using the standard ADD instruction. To make BCD arithmetic easier the 6809 provides that DAA (Decimal Adjust A) instruction. This instruction is a little different from all the other instructions that we have looked at so far in that it does nothing useful when used on its own. To be of any use the DAA instruction has to used in conjunction with the ADD instruction. Indeed it is probably better to think of the pair of instructions -

```
ADDA address
DAA
```

as a BCD ADD instruction. The DAA instruction corrects the result of an ADDA instruction so that the correct BCD result is left in A. Notice that the DAA will only correct the result of addition in the A register. Thus the DAA instruction is one instruction that makes the A and B registers different. As an example of BCD arithmetic consider adding decimal 12 to decimal 9. The BCD representation of 12 is -

```
  1  |  2
0001 | 0010
```

or $12. The BCD representation of 9 is -

```
  0  |  9
0000 | 1001
```

or $0A. Adding them together using the ADD instruction e.g.

```
LDA #$12
ADDA #$0A
```

gives the result $1C which is not the correct answer in BCD. However, following a

```
DAA
```

instruction the A register contains $21 or

```
  2  |  1
0010 |0001
```

which is the correct BCD result.

For completeness it is worth mentioning that you can write programs to add more than two BCD digits using the ADC instruction (to be introduced in Chapter Eight) to take account of the C bit. The DAA instruction adjusts not only the result but also produces a correct value for the carry in BCD arithmetic.

Adding shifts, DAA and SEX to the BASIC assembler.

None of the instructions introduced in the second half of this chapter require any modification to the BASIC assembler other than the addition of the appropriate DATA statements -

```
58 DATA SEX,-1,-1,-1,-1,&H1D
59 DATA ASRA,-1,-1,-1,-1,&H47
60 DATA ASRB,-1,-1,-1,-1,&H57
61 DATA ASR,-1,&H07,&H67,&H77,-1
62 DATA ASLA,-1,-1,-1,-1,&H48
63 DATA ASLB,-1,-1,-1,-1,&H58
64 DATA ASL,-1,&H08,&H68,&H78,-1
65 DATA MUL,-1,-1,-1,-1,&H3D
66 DATA DAA,-1,-1,-1,-1,&H19
```

Summary

1) There are three common ways of using bit patterns to represent numbers -
   i) simple binary - for positive numbers
   ii) two's complement - for negative and positive numbers
   iii) BCD - for doing decimal-like arithmetic

2) The range that can be represented by simple binary is -

for eight bits0 to 255 - represented by $00 to $FF for 16 bits0 to 65535 - represented by $0000 to $FFFF .FI .IN 0 .IN 5

3) The range that can be represented by two's complement is-

for eight bits -128..0.. + 127 - represented by $FF..$00...$7F for 16 bits -32768..0.. + 32767- represented by $FFFF..$0000..$7FFF

4) A single memory location can hold two BCD digits giving a range of 0 to 99.

5) Arithmetic on binary numbers, both simple and two's complement, can be carried out on eight-bit values using ADD, SUB and NEG. Multiplication on simple eight-bit values can be carried out using MUL and the arithmetic shift instructions can be used to multiply and divide two's complement numbers by two.

6) Sixteen bit arithmetic can be achieved using the D register and LDD, STD, ADDD and SUBD for both simple and two's complement binary numbers. Eight-bit two's complement values can be converted into 16-bit two's complement values by the use of the SEX instruction.

7) Finally BCD addition can be carried out two digits at a time using the instruction pair ADDA, DAA.

Micro projects

1) Write a short program that will add together a pair of eight-bit simple binary numbers. Assume that the first number is stored in @DATA1, the second in @DATA2 and that the result has to be stored in @ANS. Would your program be capable of adding together 200 and 50?

2) Make any modifications that are necessary to the program in question 1 so that it can add together a pair of two's complement numbers. Would your program be capable of adding together 105 and -15?

3) Write a short program that will multiply an eight-bit two's complement number by 9. (Hint: you cannot use the MUL instruction on two's complement numbers!)

4) Write a short program to subtract a 16-bit number from an eight-bit number assuming that both are two's complement representations. Assume that the first number is stored in a pair of memory locations the first of which is labelled @BIG and the second is stored in a single memory location labelled @LITTLE. How many memory locations will you need to store the answer if you do nothing more to it after the subtraction?

Chapter Seven

Branch Instructions

This chapter introduces the assembly language equivalent of the IF..THEN GOTO instruction of BASIC, However, instead of a single instruction thatcan cause a jump as a result of a range of different conditions, as with the BASIC IF, assembler uses a range of instructions each one causing a jump in response to a different condition. This range of instructions is collectively referred to as the 'branch' group or the 'conditional branches'. Once you know how to use the conditional branches the way is clear to use the assembly language equivalents of IF statements, conditional loops and FOR loops.

As these program forms are so important, you may be wondering why they have been left so until such a late chapter for discussion. The reason is that all the conditional branches make use of another register in the 6809 - the CC or 'Condition Code' register - that records certain facts about the result of the last operation. This is simple enough. The complication is that these factsare always recorded but whether they make any sense or not depends on what you are using the bit patterns to represent. For example, the CC register records whether or not the last result was negative or positive but this information is only meaningful if you are working with two's complement numbers. For this reason, a consideration of the conditional branches have been left until you have made the acquaintance of not only simple binary but two's complement and BCD representations as well. It is possible to use the branch instructions without worrying about the subtleties of what the bit pattern is representing but sooner or later such an approach will lead you to make a programming error that you have little chance of finding! The branch group of instructions are best treated as a special group by the BASIC assembler because they all use a new addressing mode, 'relative addressing', which is also described in this chapter.

Unconditional branches and relative addressing.

The 6809 has two instruction BRA (BRAnch) and LBRA (Long BRanch) that seem to do the same job as the JMP instruction in that they both transfer control to an instruction at a specified address. However, the way that the address is specified in a BRA or LBRA instruction is different and has certain advantages over the addressing modes that can be used with the JMP instruction. The addressing mode used with the BRA and LBRA instructions is known as 'relative addressing'. The form of the BRA instruction is -

BRA offset

where 'offset' is an eight-bit two's complement binary number. As the machine code for BRA is $20 and the offset is eight bits the whole BRA instruction can be stored in just two memory locations. As already mentioned, the BRA instruction behaves like the JMP instruction in that it transfers control to the machine code instruction at a particular address, The only question that remains to be answered is how the offset determines the destination address (i.e. the address that control will be transferred to). This it does in a very clever way by giving how far away, in terms of memory locations, the destination address is. To be more exact the offset specifies the number of memory locations the destination address is away from the location of the next instruction to be carried out (see fig 7.1). The reason for this use of the start of the next instruction as the position that the distance to the destination address is measured from is a result of the way that the 6809 works. If you recall, the PC or Program Counter always points at the address of the next instruction that will be obeyed. While the BRA instruction is in the process of being carried out the PC register is pointing at the first memory location of the next instruction and all that the BRA instruction does is to add the offset to this value of the PC register (see fig 7.1). If the address of the start of the BRA instruction itself is M then the destination address can be anywhere in memory from -

M + 2-128 up to M + 2 + 127

As M + 2 is the start of the next instruction and the range of an eight-bit two's complement number is -128 to + 127. Thus you cannot transfer control to any memory location using the BRA instruction. You might think that only being able to reach memory locations roughly 100 locations higher and lower than the position of the BRA instruction would be so restricting that the JMP

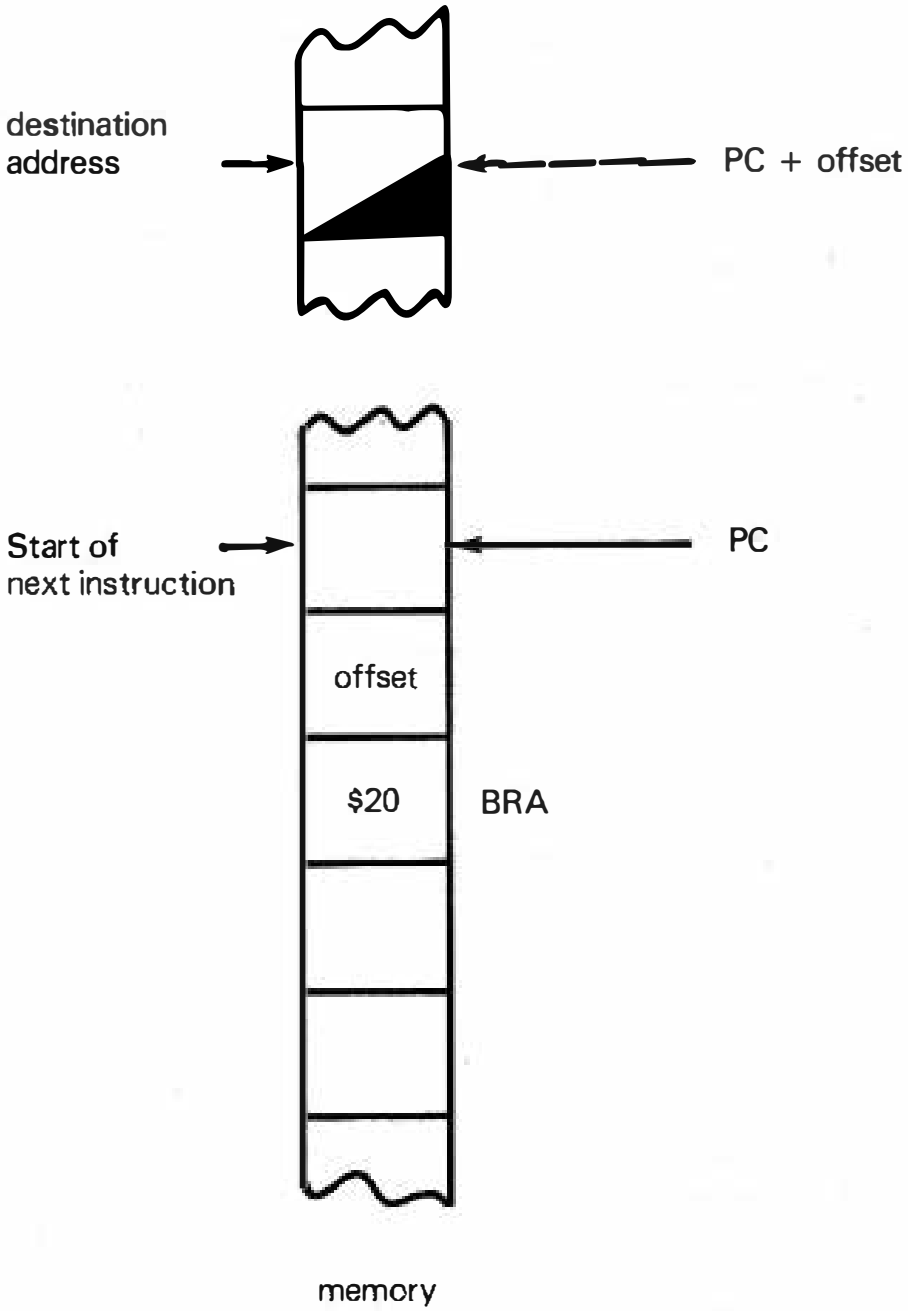


Fig 7.1 Relative addressing and BRA

instruction would always be preferred. This is not the case because, in practice, many of the destinations of jumps within a program are less than 100 memory locations away and the BRA instruction takes less memory to store and is slightly faster than the equivalent JMP instruction. These considerations of efficiency are, however, only a small part of the reason for using the BRA instruction. If you JMP to a location within a program then, when it is assembled, the address of the destination is built into the machine code. This is fine as long as the program is stored in the position in memory that it was assembled to occupy. However, as the destination of a BRA instruction is assembled into the program as an offset from the current address that the BRA instruction occupies, the BRA will work no matter where the program is loaded into memory. For example, if a BRA instruction transfers control to an instruction 10 memory locations away then the instruction will be 10 memory locations away no matter where the whole program is loaded in memory. If the program uses nothing but relative addressing then it can be loaded and run correctly anywhere in memory- that is it is 'position independent'. Position Independent Code (PIC) is a fairly advanced topic and it is best left until you have gained plenty of standard 6809 programming experience. It is, however, worth commenting on the fact that the 6809 can produce completely position independent code.

The position independence property of relative addressing is so useful that the 6809 has a second unconditional branch instruction, LBRA (Long BRAnch), which uses a 16-bit two's complement offset, allowing the destination address to be anywhere in memory. The format of the LBRA instruction is -

```
LBRA offset
```

where the machine code for LBRA is $16 and the offset occupies two memory locations in the standard two's complement format. Once again, the destination address is obtained by adding the offset to the address of the next instruction. The only difference is that the LBRA instruction takes up three memory locations and the offset is a 16-bit two's complement number. This makes the addressing range of the LBRA instruction

M+3-32768 up to M+3+32767

where M is the address of the start of the LBRA instruction.

Relative addressing and the BASIC assembler

Relative addressing is an addressing mode unique to the branch group of instructions For this reason, it makes sense to deal with it and the branches separately from the rest of the 6809 instructions and addressing modes. The easiest and most efficient way of doing this is to change subroutine 4000 to check for a mnemonic starting with B or LB. With one exception, the instruction BIT, any mnemonic that starts with B is a member of the branch group and any instruction starting with LB is a long branch. The machine codes for the branch group are kept as a separate list of DATA statements starting at 400. This list can be simpler than the list used for other machine code instructions because the branch instructions either use eight-bit or 16-bit relative addressing. The format of each 'branch' data statement can therefore be -

DATA mnemonic, code for eight-bit relative, code for 16-bit relative

and this has the added advantage of being very similar to the format of the branch table in Appendix I. Once a branch instruction has been detected then all that has to be done is to read through the table of other instructions until the ZZZ that marks the end is encountered and then search though the branch table, remembering to use the new format for the DATA statements.

```
1 REM BASIC ASSEMBLER V7.1

 400 DATA BRA,&H20,&H16
 499 DATA ZZZ,-1,-1

4009 IF (LEFT$(M$,1)="B" AND LEFT$(M$,3)<>"BIT") OR
LEFT$(M$,2)="LB" THEN GOTO4500

4500 READC$
4510 FOR K=1 TO5:READC(K):NEXT K
4520 IFC$<>"ZZZ" THEN GOTO4500
4530 IF LEFT$(M$, 1) = "L" THEN M$= RIGHT$(M$,3):BR =2 ELSE
BR = 1
4535 TYPE=BR
4540 READC$
4550 FORK=1TO2:READC(K):NEXTK
4560 IFC$="ZZZ"THENI=I+1:ER=1:RETURN
4570 IFC$=M$THENRETURN
4580 GOTO4540
```

Lines 400 and 499 are the only DATA statements in the branch list so far (499 marks the end of the list with ZZZ). Line4009 detects the the mnemonic is one of the branch group and passes control to the routine at 4500 to 4580. This routine first skips over the rest of the instruction table and then reads the branch table to find the matching mnemonic. Line 4530 test to see if the instruction is a branch or a long branch. If it is a long branch then the L is removed from the front of the mnemonic and BR is set to 2. If it is a normal branch then BR is set to 1. The variable TYPE is then set to the same value as BR, because once the matching mnemonic in the table is found the code for a branch will be in C(1) and for a long branch in C(2). It should be noticed that this doesn't imply any connection between relative addressing and address modes corresponding to TYPE= 1 and TYPE =2 (immediate and direct) it is simply a programming convenience.

The problem of detecting branch instructions and finding the correct machine code is fairly easy but what about handling relative addressing? In machine code the destination address is specified as an offset from the address of the next instruction but is there any need to inflict this difficult calculation on the assembly language programmer? The answer is clearly no! The assembler should allow the programmer to specify addresses in the usual way either by writing the actual address or by using a label, and then it should produce the correct machine code by calculating the necessary offset. This means that instructions like -

```
BRA @LOOP
LBRA $BF00
```

are perfectly correct assembly language. To a certain extent, this normal use of addresses hides the fact that BRA and LBRA are using relative addressing. However, you will soon be reminded of the fact if, for example, @LOOP is too far away to be reached by an eight-bit offset!

The modification to the BASIC assembler has to convert the address value in A to a relative address if the instruction is a branch. Detecting that the address field belongs to a branch instruction is simply a matter of testing the variable BR for a value greater than 0 (BR= 1 is a branch, BR =2 is a long branch). Converting the address to relative form is a little more complicated. If the branch is to transfer control to the address stored in A and the address at

which the branch instruction will be stored is in P, then the offset can be calculated using -

OF = A - 2 - P

for a branch and

OF = A - 3 - P

for a long branch.

Recalling that the value of BR for branches and long branches is 1 and 2 respectively, you should be able to see that the offset is equal to -

OF = A - BR - 1 -P

in both cases. If the offset turns out to be positive then there is nohing else to be done apart from POKEing 1 memory location for an eight bit relative address and 2 memory locations for a 16-bit relative address. If the offset is negative then it has to be converted to the correct two's complement form before being POKEd into memory. For an eight-bit relative address this can be done by -

256-OF

and for a 16-bit relative address by

65536-OF

Putting all this together gives the following modification

```
5024 IF BR>0 THEN GOTO 5700

5700 IF PASS=1 THEN A=0:RETURN
5705 OF= A-BR-1-P
5710 IF BR=1 AND (OF< -128 OR OF> 127) THEN
ER=4:I=I+1:GOTO9000
5720 IF OF> =0 THEN A= OF:RETURN
5730 IF BR=2 THEN GOTO 5760
5740 A=256-OF
```

```
5750 RETURN
5760 A=65536-OF
5770 RETURN

6021 IF BR=2 THEN TYPE=4
6060 TYPE=0:BR=0
```

Line 5024 detects a branch instruction and hence a relative address and transfers control to a new routine at 5700. This calculates the offset, including conversion to two's complement if necessary, as described earlier. Line 5710 checks that an eight-bit relative address is in the correct range.

Conditional branches - the Condition Code register

The BRA and LBRA instructions always transfer control to their destination addresses - in this sense they form another two assembly language equivalents of the BASIC GOTO instruction. The conditional branches are similar to the BRA and LBRA instruction in that they use the same addressing modes - eight-bit and 16-bit relative addressing - respectively - but they do not always transfer control to their destination addresses. In this sense the conditional branches form the assembly language equivalent of the BASIC IF condition THEN GOTO xxx. However, in this particular case the range of conditions that can be tested is very restricted. Also, whether or not the branch is taken depends on the result of a previous operation rather than any calculation built into the branch instruction itself. For example, the BEQ (Branch EQual) instruction will transfer control to its destination address if the result of the last operation was zero. So if an addition or subtraction instruction is followed by a BEQ it is the result of this prior arithmetic which is tested to see if it was 0 to decide whether or not the branch is taken.

This idea of looking back at the result of the last operation to decide the outcome of a branch instruction is perhaps easier to appreciate once you know how it works. The CC or Condition Code register has already been mentioned in earlier chapters as the place where the C or Carry bit is stored. In fact it is an eight bit register in which each of the bits has a special name and a special function. For example, one of the bits in the CC register, b2, is called the Z or Zero bit and this is set to 1 if the result of the last operation was zero and a 0 otherwise. Notice that the Z bit, like the C bit, is changed by the

previous operations even if it is not going to be used in anyway later on. In this sense the CC register really does monitor the 'condition' of the 6809 by continually recording information about previous results.

Not all of the eight condition bits within the CC register are of interest to us at this stage. Of the eight bits only five are concerned with the outcomes of operations on data and so only these five are described below:

The Condition Code Register

b7	b6	b5	b4	b3	b2	b1	b0
.	.	H	.	N	Z	V	C

where '.' indicates that the purpose of the bit will be described in Chapter Ten.

The H bit

The H or Half carry bit is used by the DAA instruction to do BCD arithmetic. (See Chapter Six). Normally this bit isn't of any direct interest or use to the assembly language programmer. The best policy is to accept that the DAA instruction uses it and then forget about it.

The N bit

The N or Negative bit is always equal to b7 of the last result. Obviously if you are working with simple binary numbers all that the N bit will indicate is whether or not the last result was equal to or bigger than $80 and this is not very useful. However, if you are working with two's complement numbers, b7 is 1 if the number is negative and 0 if it is positive. Thus for two's complement numbers the N bit indicates whether or not the last result was negative or not.

The Z bit

The Z or Zero bit has already been discussed briefly. It is 1 if the previous result was zero and 0 otherwise.

The V bit

The V or oVerflow bit is the most complicated of all the condition code bits. It is 1 if the last operation caused a two's complement overflow and 0 otherwise. Of course if you are not using the two's complement representation then the V bit's value makes no sense at all!

The C bit

The C or Carry bit has already been described in Chapters Five and Six in connection with the logical and arithmetic shift operations. Its most common use is as a carry or borrow in multiple precision arithmetic and this is explained fully in Chapter Eight. However, it is also used to test for overflow in simple binary arithmetic and as a way of discovering the value of the bit shifted out as a result of a shift instruction. In its role as a carry bit used to detect overflow it is best thought of as a ninth bit produced as a result of an eight-bit addition or subtraction. You should be able to see that following simple eight-bit binary arithmetic the C bit should be 0 if the result can be represented in eight bits. In the same way it can be thought of as a 17th bit as a result of 16-bit addition or subtraction. Once again if the result can be correctly represented in 16 bits there should be no carry and the C bit should be 0.

Notice that the four condition code bits N, Z, V and C that are used by the conditional branches fall into two groups. The N and Z bits both reflect a property of a single number or bit pattern - i.e. is it negative or zero respectively. However, the V and C bits reflect an aspect of the result of an operation - i.e that an overflow or a carry occurred. For this and various other reasons, not every instruction changes all, or even any, of the condition codes. It is obviously very important to know which of the condition code bits any given instruction will change but to list each one here would take too much space. A complete list of which bits each instruction affects is given as part of Appendix I and this should be used for reference. Fortunately, it is usually easy to work out which instructions do and do not change the condition codes. For example, instructions which load registers set the N and Z bits so that tests can be made on the value loaded. The C bit is left unaltered (it is difficult to see how a 'carry' can arise when you are not doing any arithmetic!) and the V bit is zeroed because no overflow has occurred. However, until you are familiar with 6809 assembler the best policy is to look up in Appendix I which condition codes are affected by the instructions prior to making use of the condition codes.

Setting the condition codes directly - ANDCC and ORCC.

The condition code bits are used by the conditional branch instructions to determine whether or not the branch should be taken. Before discussing the details of each conditional branch instruction it is worth examining ways of intentionally setting the condition code bits.

There are two instructions that allow the assembly language programmer to alter the condition code register directly - ANDCC and ORCC. Both instructions can only be used with immediate addressing and the action of -

```
ANDCC #data
```

is to form the AND of the current contents of the CC register and the eight bit value 'data' leaving the result in the CC register. The action of

```
ORCC #data
```

is to form the OR or the current contents of the CC register and the eight bit value 'data' leaving the result in the CC register. If you followed the discussion of bit manipulation in Chapter Five you will realise that ANDCC and ORCC allow you to set any bit in the CC register to either 0 or 1 without affecting the current state of any of the other bits. For example, if you want to set the C bit to zero (sometimes referred to as 'clearing the carry') then use-

```
ANDCC #$FE
```

If you want to set the C bit to 1 then use -

```
ORCC #$01
```

Although you can change any of the bits in the CC register in practice it is only the C bit that is ever changed directly in this way. Generally before any ROL or ROR instruction the C bit should be initialised to either 0 or 1 as appropriate. (This is also the case when using multiple precision arithmetic with the ADC and SBC instructions to be explained in Chapter Eight.)

The simple conditional branches

The conditional branches are best thought of as being made up of three different groups:

> he simple conditional branches that each test a single condition code bit.
>
> The signed conditional branch instructions that are used with two's complement numbers.
>
> The unsigned conditional branches that are used with simple binary numbers.

In this section the simple conditional branch instructions will be described.

As the simple conditional branch instruction each test one of N, Z, V and C in the condition code register you might expect that there would be exactly four such instructions. In fact there are eight simple condition branches because each condition code bit is associated with a pair of branches - one that is taken if the bit is 0 and one that is taken if the bit is 1. The complete set of simple branch instructions is -

bit	branch taken if 0	branch taken if 1
N	BPL - Branch PLus	BMI - Branch MInus
Z	BNE - Branch Not Equal	BEQ - Branch EQual
V	BVC - Branch V Clear	BVS - Branch V Set
C	BCC - Branch C Clear	BCS - Branch C Set

Each of these instructions can only be used with eight-bit relative addressing. If the destination address is outside the range of eight-bit relative addressing then each of the simple branch instructions is available in a long branch form, e.g. BPL is also available as LBPL (Long Branch Positive).

The way that each of the above simple conditional branches behaves is easy enough to understand. What is not so clear is that way that each one might be used as part of a program. It is perhaps worth dealing with each pair in turn.

BPL and BMI - telling positive from negative

As already described the N bit is equal to b7 of the last result that affected the condition codes. Testing the state of this bit can be used to tell the difference between a two's complement positive or negative result. In the following example,

```
LDA #$05
SUBA #$08
BMI @MINUS
```

the branch to @MINUS will be taken because the result of the subtraction is negative. Notice that as far as BPL and BMI are concerned 0 is a positive number. Of course, there is nothing stopping you from using BPL and BMI when the numbers involved in the arithmetic are simple binary. However, the interpretation of the branch as a test for positive and negative numbers then becomes a nonsense. There is one problem that can occur with the use of BPL and BMI. If during the course of a two's complement calculation the result becomes invalid due to an overflow, the BMI or BPL will still be selected according to the value of b7 of the result but this result may not be what you expect it to be. In other words, it is possible to take a correct branch from an incorrect answer! For example, consider -

```
LDA #$7F
ADDA #$2
BPL@PLUS
```

$7F is the two's complement representation of 127. Adding $2 to this should give the result 129 and so the BPL should transfer control to @PLUS. However, 129 is outside the positive range of an eight-bit two's complement number and so the result is actually $81 which is the two's complement representation of -128 and the branch to @PLUS is therefore not taken and control passes to whatever instruction follows the BPL. This problem of branching or not branching on an invalid result is a problem to be found in other branch instructions.

BEQ and BNE - testing for zero

There are no complications in using the BEQ and BNE instructions. If the result of the last operation was zero then a BEQ branch will be taken. If the result was anything else other than zero then the BNE branch will be taken. The interesting and useful thing about the BNE and BEQ pair is that there is no need to worry about number representation. Perhaps the best way to

understand this is that a BEQ will be taken if all eight bits of the results bit pattern are 0. This all 0 pattern can arise in a number of ways. For example, if the bit patterns are being used as two's complement numbers then -

```
LDA #$10
SUB #$10
BEQ@ZERO
```

The result of the SUB instruction is $0 and so the BEQ transfers control to @ZERO. A less obvious example is -

```
LDA #$F0
ANDA #$0F
BEQ @ZERO
```

The result of ANDing $F0 with $0F is $00 and so the branch to @ZERO is taken.

BVC and BVS - testing for overflow

The instructions BVC and BVS test whether or not the last operation resulted in a two's complement overflow. The V bit is so specialised that there is really only one use for the BVC and BVS instructions and that is to test for a valid result at the end of any two's complement arithmetic. For example,

```
LDA #$7F
ADDA #$02
BVS @ERROR
```

results in control being transferred to @ERROR as, in two's complement, $7F is 127 and adding 2 takes the result outside the valid range so overflow occurs and the V bit is set. You can use BVS and BVC in this way to call or skip an error handling routine. The problem of detecting overflow in multiple precision arithmetic (see Chapter Eight) is easily solved by testing for overflow after the last addition or subtraction.

BCC and BCS - testing for simple binary overflow

BCC and BCS are used following simple binary arithmetic in much the same way that BVC and BVS are following two's complement arithmetic. In other words following simple binary arithmetic there should be no carry if the result is valid. For example,

```
LDA #$FE
ADDA #$02
BCS @ERROR
```

results in the branch to @ERROR being taken because $FE added to $02 gives the result $01 and a carry. There are other uses for BCC and BCS apart from detecting simple binary overflow. By using LSL and LSR instructions it is possible to shift any bit into the C bit and then use the BCS and BCC instruction to take a branch depending on whether the bit was 0 or 1. By detecting simple binary overflow following a subtract instruction it is possible to tell that the value in the register was less than the value in the memory - but more of this later.

The signed conditional branches

The signed conditional branches are concerned with comparing the relative magnitudes of 2 two's complement numbers. In this sense they come closest to the BASIC conditional tests <, >, <= and >=. Indeed there is a signed conditional branch for each of these relations and, if BEQ and BNE are included, for = and <> as well. The signed conditional branches are -

test	branch if true	branch if false
r> m	BGT-Branch Greater Than	BLE-Branch less than or equal
r>= m	BGE-Branch Greater than or Equal	BLT-Branch Less Than
r= m	BEQ-Branch Equal	BNE-Branch Not Equal

where the branches are to be used following a subtract instruction, r is the contents of the register and m the contents of the memory. The relationships in the first column are to be interpreted as two's complement relationships. That is, in eight-bit two's complement representation $80 is less than $7F as $80 is -128 and $7F is +127. Consider, for example -

```
LDA #$50
SUBA #$24
BGT @BIGGER
```

The branch to @BIGGER is taken because $50 is greater than $24. (In this case r, the value in the register, is$50 and m, the value in the memory, is$24.) This is all there is to using the signed conditional branches.

If you subtract two numbers, one stored in a register and one in memory (or immediate data) then you can use any of the above branch instructions as long as you remember to interpret the numbers as two's complement numbers. You might think that two's complement overflow might be a problem with the signed conditional branches. In other words, do the signed branches work when a two's complement subtraction results in an invalid result? The surprising answer is that they do! Even when the result of the subtraction is invalid, i.e. a BVS is taken indicating an overflow, the signed conditional branches still work correctly. The reason for this is that the signed conditional branches can deduce the sign that the correct result would have had if it could have been worked out correctly! Consider, for example-

```
LDA #$80
SUBA #$01
BLE @SMALL
```

In this case the result of the subtraction should be -129 (because $80 is -128 in two's complement and subtracting 1 gives -129). However, because -129 is outside the range of representation of eight-bit two's complement arithmetic, the result is actually $7F or +127 which is incorrect (and the V bit is 1 to indicate overflow) but the branch to @SMALL is correctly (-128 is less than 1) taken. It is worth remembering that (and trying to understand why) the signed conditional branches will work even if the result of the subtract is invalid.

The unsigned conditional branches

The unsigned conditional branches are the simple binary equivalents of the signed conditional branches. In other words they can be used to compare two simple binary values. The unsigned conditional branches are -

test	branch if true	branch if false
r> m	BHI-Branch HIgher	BLS-Branch Lower or Same
r> =m	BHS-Branch Higher or Same	BLO-Branch LOwer
r= m	BEQ-Branch EQual	BNE-Branch Not Equal

Once again the branch instruction have to follow a subtract operation, *r* is the numbers stored in the register and m is the number stored in the memory location (or immediate data), for example -

```
LDA #$80
SUBA #$7F
BHI @HIGHER
```

In simple binary $80 is 128 and $7F is 127 so the result is 1 and the contents of the register are larger or higher than the memory location and hence the branch to @HIGHER is taken. In the same way that the signed conditional branches will branch correctly on an invalid result so will the unsigned conditional branches. Consider, for example -

```
LDA #$7F
SUBA #$80
BLO @LOWER
```

The result of 127 minus 128 is -1 and this cannot be represented in simple binary (no negative number can be represented in simple binary!) but even though the result is incorrect ($FF) the branch to @LOWER is still correctly taken.

It is worth pointing out that BLO is identical to the BCS instruction and the BHS instruction is identical to the BCC instruction. The additional mnemonics BLO and BHS are used mainly for convenience and to distinguish the uses of the BCS and BCC instruction that are concerned with testing the value of bit shifted into the carry during logical shifts etc and applications of the C bit that involve arithmetic. It is important not to confuse the signed and unsigned conditional branches. If you are using two's complement representation, i.e. if you are using negative numbers, then use the signed

conditional branches. If you are not using negative numbers then you are free to choose a simple binary representation and the unsigned conditional branches.

Adding conditional branches to the BASIC assembler

Adding the conditional branches to the BASIC assembler is a matter of adding the appropriate DATA statements to the separate branch list starting at -400. It seems worth also adding the DATA statements for the instructions that will be described in the rest of the chapter at this point to avoid generating too many versions of the BASIC assembler.

```
1 REM BASIC ASSEMBLER V7.2

67 DATA CMPA,&H81,&H91,&HA1,&HB1,-1
68 DATA CMPB,&HC1,&HD1,&HE1,&HF1,-1
69 DATA CMPD,&H1083,&H1093,&H10A3,&H10B3,-1
70 DATA BITA,&H85,&H95,&HA5,&HB5,-1
71 DATA BITB,&HC5,&HD5,&HE5,&HF5,-1
72 DATA ANDCC,&H1C,-1,-1,-1,-1
73 DATA ORCC,&H1A,-1,-1,-1,-1
74 DATA TSTA,-1,-1,-1,-1,&H4D
75 DATA TSTB,-1,-1,-1,-1,&H5D
76 DATA TST,-1,&H0D,&H6D,&H7D,-1

401 DATA BCC,&H24,&H1024
402 DATA BCS,&H25,&H1025
403 DATA BEQ,&H27,&H1027
404 DATA BGE,&H2C,&H102C
405 DATA BGT,&H2E,&H102E
406 DATA BHI,&H22,&H1022
407 DATA BHS,&H24,&H1024
408 DATA BLE,&H2F,&H102F
409 DATA BLO,&H25,&H1025
410 DATA BLS,&H23,&H1023
411 DATA BLT,&H2D,&H102D
412 DATA BMI,&H2B,&H102B
413 DATA BNE,&H26,&H1026
414 DATA BPL,&H2A,&H102A
```

```
415 DATA BSR,&H8D,&H17
416 DATA BVC,&H28,&H1028
417 DATA BVS,&H29,&H 1029
```

The only other modification to be made to the assembler is to take account of the fact that most of the long branch instructions use two memory locations for their machine code. This use of two memory locations to code an instruction is a way of getting round the limitation of a single memory location only being able to code 256 different instructions. The 6809 uses two codes $10 and $11 as special markers to indicate that there is a second part to the code stored in the next memory location. In other words there are two groups of instructions that use two memory locations to store their code, a group called 'page two instructions' that start $10 and a group called 'page three instructions' that begin $11. Apart from taking up more memory and being a little slower, instructions that need more than one memory location for their code are only a problem to the assembler. However, this double length form of the instruction can be dealt with in the BASIC assembler very easily -

```
6010 IF C(TYPE)< 256 THEN POKE P,C(TYPE)
6011 IF C(TYPE)>255 THEN POKE P,INT(C(TYPE)/256): POKE
P+1,C(TYPE)-INT(C(TYPE)/256)*256:P=P+1
```

Testing without changing - CMP, TST and BIT

The conditional branches do provide us with a way of transferring control to some part of a program depending on the result of comparing two numbers in a way that is not unlike the BASIC IF statement. However, to compare two values it is necessary to subtract one from the other as shown in the many examples above. This subtraction produces a result (i.e. the difference between the two numbers!) that may not be required by the program. For example, it may be that you want to use the BEQ instruction to test when the A register is equal to a particular value. The obvious and direct way of doing this would be -

```
SUBA #$10
BEQ  @LOOP
```

where the contents of the A register are compared to $10 by having $10 subtracted and then a BEQ takes its branch if the result was zero. This method works but it destroys the current value in the A register, replacing it by the

result of the subtraction, and it might be that if A isn't equal to $10 then the current value is used by later parts of the program. One solution would be to save the A register before the subtraction and re-load it with its old value after the subtraction and the branch but this is a very inefficient way of comparing two values.

The correct solution is to use the CMP instruction which will carry out the subtraction and set the condition codes in exactly the same way as a SUB command but will not save the result. In other words, following a CMPA #$10 command the condition codes are set as if $10 had been subtracted from the A register but the contents of the A register are left unchanged. Now the problem of comparing two numbers is completely solved and the previous example can be written -

```
CMPA #$10
BEQ @LOOP
```

The branch will be taken if the A register is equal to $10 but the contents of the A register remain unaltered no matter what happens.

The CMP instruction has another advantage over the SUB instruction in that it is available as

```
CMPA address
CMPB address
```

and

```
CMPD address
```

The final form allows you to compare the contents of the D register (see Chapter Six) with a 16-bit value and branch on the result. For example-

```
CMPD #$0172
BLT @LESS
```

will take the branch to @LESS if the contents of D are less than $0172 in two's complement representation.

The problem of testing to see if a value is zero, positive or negative is so common that there is a special instruction for just this purpose. The TST instruction has exactly the same effect as a CMP to zero but it can be used to test the value stored in a memory location directly. TST comes in three forms. The first two are

```
TSTA
```

and

```
TSTB
```

which have the same effect (but take less memory) than

```
CMPA #0
```

and

```
CMPB #0
```

The third form

```
TST address
```

effectively subtracts zero from the memory location at address and sets the condition code bits accordingly. TST can be used with direct and extended addressing.

In the same way that the CMP operation performs a subtraction without altering the contents of the register involved, the BIT (BIt Test) instruction performs the AND operation with out changing the register. A typical application of the BIT instruction is to discover when two bit patterns, whatever they represent, are the same or not. For example, suppose you wanted to know if the value stored in the A register had b4 set to 1 irrespective of the setting of the other bits. Then you could use -

```
BITA #$10
BNE @SET
```

because the result of ANDing anything with $10 can only be non-zero if b4 is set to 1 - try it! As with the CMP instruction, following BITA the contents of the A register are unchanged. The BIT instruction has only two forms-

```
BITA address
```

and

```
BITB address
```

In other words, there is NO 16-bit BITD form.

Adding CMP and BIT to the BASIC assembler is simply a matter of adding their DATA statements and this has already been done in V7.2 given above.

BSR and LBSR

There are two remaining unconditional branches to be described. To a certain extent this is because they are not really needed unless you are trying to write position independent programs (see earlier in this chapter). The BSR (Branch to SubRoutine) instruction behaves exactly like the JSR instruction in that it transfers control to a subroutine except that, like all the branches, it uses relative addressing. If the subroutine is out of the range of eight-bit relative addressing then you can use LBSR (Long Branch to SubRoutine) which, using a 16-bit offset, can reach a subroutine anywhere in memory. It is good assembly language programming practice to use BSR and LBSR to jump to subroutines within a program that you are writing but JSR to jump to a subroutine external to your program - in the BASIC ROM say. The reason for this is discussed more fully in the last chapter but, essentially, subroutines within your program will move their location if you move your program and so to produce position independent you should use relative addressing. However, subroutines that are outside your program generally don't move when your program does and so relative addressing would be inappropriate in this case. BSR and LBSR have already been added to the BASIC assembler.

Thinking BASIC - IF, conditional loops and FOR loops

Most of the work in a BASIC program is done using IF statements, conditional loops and FOR loops. Now that we know about the conditional branches it is possible to see how these familiar statements can be made up in assembler.

The only form of the IF statement that has an easy translation into assembly language is IF condition THEN GOTO where the 'condition' is a simple relation between two values. For example, the BASIC -

```
100 IF ANS> CONST THEN GOTO 300
```

is similar to the assembler -

```
LDA  @ANS
CMPA @CONST
BGT  @SKIP
```

where @ANS and @CONST are memory locations holding the two values to be compared and @SKIP labels an instruction further down the program. Notice that as BGT has been used the assumption is that the two values are two's complement numbers in the range -128 to +127. If boh values cannot become negative then the positive range could be extended to +255 but the branch instruction would then have to be changed to BHI @SKIP. Once again it is important to be clear what the bit patterns are being used to represent.

The conditional loop in BASIC is simply a GOTO statement arranged to form a loop and an IF statement to transfer control out of the loop when a condition is satisfied. For example -

```
10 COUNT=0
20 COUNT=COUNT+10
30 IF COUNT=100 THEN GOTO 1000
40 GOTO 20
```

is a simple conditional loop that adds 10 to COUNT until it is equal to 100. The assembly language equivalent of this would be -

```
        CLRA
@LOOP   ADDA    #$0A
        CMPA    #$94
        BEQ     @FINISH
        BRA     @LOOP
```

Notice the way that the BRA instruction is used to form a loop in the same way that a GOTO would be in BASIC. Also notice that as the BEQ instruction works with any representation the form of the program remains the same even if you are working with two's complement numbers. In this case 100 is outside the range of eight-bit two's complement numbers so a simple binary representation is used.

Following the conditional loop the assembly language equivalent of the FOR loop is particularly easy. For example, the assembly language equivalent of -

```
FOR I=1 TO 10
. . .
NEXT I
```

is

```
        LDB  #$01
@FOR1   'other
        assembly
        language'
        . . .
        INCB
        CMPB #$0A
        BLE @ FOR 1
```

This form of the assembly language FOR loop can be modified to include a negative step by changing INCB to DECB and to allow for STEP sizes larger than 1 by using ADDB #@STEP. For all its flexibility, however, the needs of assembly language programming can often be met by something a lot simpler than this. Very often all that is required is to repeat a section of program a number of times. For example, suppose you wanted to shift the contents of the A register to the left four times you could use -

```
LDB #$04
@LOOP     LSLA
DECB
BNE@LOOP
```

The way that this works is to load the B register with the number of times you want to repeat the instruction inside the loop and then DECrement the B register each time through. When the contents of the B register reach 0 the BNE is not taken and the loop ends. Notice that using this method of counting down to 0 there is no need to use a CMP instruction to discover when the loop is at an end because the Z bit is automatically set by the DEC instruction.

There are so many ways of using conditional branches to construct equivalents of BASIC statements that there isn't enough room to give a complete list. In any case it is one of the features of assembly language programming that exactly how something is best done depends very much on the requirements of the rest of the program. In short, you should understand assembly language well enough to make up your own equivalents to fit in with the rest of your program.

Summary

1) The condition code register is used to record certain features of the result of 6809 operations. Only four condition code bits that are used by the conditional branches the N, V, Z and C bits. The condition code bits can also be modified directly using the ANDCC and ORCC instructions.

2) The relative addressing mode is used with the branch group of instructions. Roughly speaking, a relative address specifies a memory location in terms of how far away, in terms of number of memory location it is from the current position within the program.

3) The branch instructions come in two forms, the branches and the long branches using eight and 16-bit relative addressing respectively. The conditional branches fall into three groups -

i) the simple branches that test a single bit in the CC register
ii) the signed conditional branches that work with two's complement numbers
iii) the unsigned conditional branches that work with simple binary numbers

4) The CMP, TST and BIT instructions can be used to compare numbers and bit patterns without storing any results and so without modifying the contents of the registers. As well as the usual A and B register form of the CMP instruction, there is a very useful 16-bit CMPD form. The TST instruction can be used to test for a zero, positive or negative value stored in the A or B registers or in a memory location.

Micro projects

1) Using ANDCC and ORCC set the Z bit to 1 and the C bit to 0.

2) Write a short program that is the assembly language equivalent of -

```
FOR I=10 TO 1 STEP -1
....
NEXT I
```

Use the B register as the index variable I.

3) Write a program that adds two simple binary numbers together and calls the machine code subroutine @RESULT if the answer is valid and @ERROR is not. You do not have to write the subroutines @RESULT and @ERROR.

4) Change the program in (3) to add a pair of two's complement numbers.

Chapter Eight

Using the Dragon from Assembler

Although there have been plenty of short examples of 6809 assembly language programs throughout this book, this is the first chapter where the examples are of any size or usefulness. To start the chapter a few simple routines are developed showing how data can be manipulated using the instructions introduced in the last few chapters. The largest example in this chapter is a simple video game. The reason for choosing a video game is that it is easy to understand the objectives of the program and easy to see if they have been met. Also, moving an object around the screen provides an excellent example of how careful you have to be in handling numbers and doing arithmetic. All in all a video game makes an excellent example, illustrating both the advantages and the difficulties of assembly language programming!

However, before we can move on to the examples proper, something is going to have to be done to make the BASIC assembler easier to use. Although it is has been good enough to assemble the short program examples given so far it is not really suitable for developing programs from scratch or even for typing in examples longer than a few lines. It's not that the assembler won't cope with the problems of assembling the programs, it's just that it provides none of the niceties of of any sort of editing, of saving and loading assembly language programs on tape, or a reasonably clear and useful listing.

Making the BASIC assembler friendly

Changing the BASIC assembler to include editing and other features to make it easier to use is not difficult but it is the largest single change so far. None of the methods used are in away new or difficult and rather than give a

detailed explanation of the whole program, a table giving the purpose of each subroutine is given at the end of the listing.

```
1 REM BASIC ASSEMBLER V8.1

5 CLEAR 2000,&H6FFF


590 I= T:GOSUB 1980
600 LC=0:GOTO 515


1000 DIM A$(150),C$(5),T$(50),T(50)
1980 PRINT "PRESS ANY KEY TO CONTINUE";
1990 IF INKEY$="" THEN GOTO 1990

2000 CLS
2010 PRINT@66,"BASIC ASSEMBLER"
2020 PRINT
2030 PRINTTAB(10);"SELECT ONE OF"
2040 PRINT
2050 PRINTTAB(8); "INPUT/EDIT.....1"
2060 PRINTTAB(8);"ASSEMBLE.....2"
2070 PRINTTAB(8);"SAVE ONTAPE....3"
2080 PRINTTAB(8);"LOAD FROMTAPE..4"
2090 PRINTTAB(8);"EXECPROGRAM....5"
2100 INPUT ACTION
2110 IFACTION< 1 ORACTION>5THEN GOTO2000
2120 ON ACTION GOTO 2200,2800,2850,2920,2990

2200 CLS
2210 PRINT@76,"EDIT"
2220 PRINT
2230 PRINTTAB(10);"SELECT ONEOF"
2240 PRINT
2250 PRINTTAB(8);"LIST PROGRAM....1"
2260 PRINTTAB(8);"LISTTO PRINTER.2"
2270 PRINTTAB(8);"ADDTOPROGRAM..3"
2280 PRINTTAB(8);"DELETE LINES....4"
2290 INPUTED
```

```
2300 IFED<1ORED>4THENGOTO2000
2310 ONEDGOTO2400,2400,2500,2700


2400 CLS
2405 IFI=0THENGOTO1980
2410 FORK=1TOI
2420 IF ED=1 THEN PRINT K;":";TAB(4);A$(K) ELSE
PRIN T#-2,K;":";TAB(4);A$(K)
2430 NEXTK
2440 GOTO1980


2500 IFI=0THENGOTO2620
2505 INPUT"ADDLINESFOLLOWINGLINENUMBER";LN
2510 IFLN>=ITHENLN=I:GOTO2620
2520 INPUT"NUMBEROFLINESTOINSERT";IN
2530 IFI+IN>150THENPRINT"TOOMANY":GOTO2000
2540 FORK=ITOLN+1STEP-1
2550 A$(K+IN)=A$(K)
2560 NEXTK
2570 FORK=LN+1TOLN+IN
2580 PRINTK;":";TAB(4);
2590 LINEINPUTA$(K)
2600 NEXTK
2605 I=I+IN
2610 GOTO1980
2620 PRINT"TYPE END TO FINISH"
2630 K=I+1
2640 PRINTK;":";TAB(4);
2650 LINEINPUTL$
2660 IFLEFT$(L$,3)="END"THENGOTO1980
2670 I=K:A$(I)=L$
2680 GOTO2630


2700 INPUT"FIRST LINE TO DELETE";FL
2710 INPUT"LAST LINE TO DELETE";LL
2720 IFLL<FLTHENPRINT"NOT DELETED":GOTO1980
2730 FORK=LL+1TOI
2740 A$(FL+K-LL-1)=A$(K)
2750 NEXTK
```

```
2770 GOTO 1980
2760 I=I-(LL-FL+1):PRINT "DELETED"

2800 INPUT "SCREEN (0) OR PRINTER(1)";PRT
2810 T=I
2820 PRT=PRT*2
2830 RETURN

2850 INPUT"FILE NAME";F$
2860 PRINT"PRESS PLAY AND RECORD"
2870 PRINT"PRESS ANY KEY WHEN READY"
2880 IF INKEY$="" THEN GOTO2880
2890 OPEN"O",#-1,F$
2900 FOR K=1 TOI:PRINT #-1,A$(K):NEXTK
2910 CLOSE #-1:GOTO1980

2920 INPUT"FILE NAME";F$
2930 PRINT"PRESS PLAY"
2940 OPEN"I",#-1,F$
2950 I=0
2960 IF EOF(-1) THEN CLOSE #-1:GOTO 1980
2970 I=I+1:INPUT#-1,A$(I)
2980 GOTO2960

2990 CLS:EXEC &H7000
2995 GOTO 1980

6000 IF C(TYPE)=-1 THEN ERR=5:GOTO9000
6010 IF PASS=1 THEN GOTO6200
6020 PRINT#-PRT,RIGHT$("  "+HEX$(P),4);TAB(5);
6030 PRINT#-PRT,HEX$(C(TYPE));TAB(8);
6040 IF TYPE<>5 THEN PRINT#-PRT, HEX$(A);
6050 PRINT#-PRT,TAB(15);A$(I)
6200 IF C(TYPE)<256 THEN POKE P,C(TYPE)
6210  IF C(TYPE)>255 THEN POKE P,INT(C(TYPE)/256): POKE
P+1,C(TYPE)-INT(C(TYPE)/256)*256:P=P+1
6220 P=P+1
6230 IF BR=2 THEN TYPE=4
6240 IF TYPE=1 AND RIGHT$(M$,1)="D" THEN TYPE=4
6250 IF TYPE=5 THEN RETURN
6260 IF TYPE=2 OR TYPE=1 THEN POKE P,A
```

```
6270 IF TYPE=4 THEN POKE P,INT(A/256):P=P+1: POKE
P,A-INT(A/256)*256
6280 TYPE=0
6290 BR=0
6300 P=P+1
6310 RETURN

6500 IFPS<>1THENGOTO6540
6510 IFPASS=1THENT(LC)=A
6520 IFPASS=2THENPRINT#-PRT,TAB(15);A$(I)
6530 RETURN
6540 IFPS<>2THENGOTO6570
6545 IFPASS=2THENPRINT#-PRT,HEX$(P);
6550 P=P+A
6560 IFPASS=2THENGOTO6520
6570 IFPS<>3THENGOTO6650
6580 A=A-INT(A/256)*256
6590 IFPASS=1THENGOTO6620
6600 PRINT#-PRT,HEX$(P);TAB(5);HEX$(A);
6610 PRINT#-PRT,TAB(15);A$(I)
6620 POKEP,A
6630 P=P+1
6640 RETURN
6650 IFPS<>4THENRETURN
6660 IFPASS=1THENGOTO6710
6670 LB=A-INT(A/256)*256
6680 HB=INT(A/256)
6690 PRINT#-PRT,HEX$(P);TAB(5);HEX$(HB);TAB(8);HEX$(LB);
6700 PRINT#-PRT,TAB(15);A$(I)
6705 POKEP,HB:POKEP+1,LB
6710 P=P+2
6720 RETURN

9000 PRINT#-PRT,"ERROR--";ER;"*** AT LINE";I
9010 RETURN
```

routine	purpose
1980-1990	press any key to continue entry to 2000
2000-2120	main menu and selection

2200 2310	editing menu and selection
2400-2440	list to screen or printer
2500-2680	add/insert lines to program
2700-2770	delete lines from program
2800-2830	assemble program
2850-2910	save to tape
2920-2980	load from tape
2990-2995	EXEC program
6000-6310	new version of subroutine 6000 to list assembly
6500-6720	new version of subroutine 6500 to list pseudoops
9000-9020	new version of error handler

Subroutine 6000 is almost completely new and it is better to delete the old version and type the new one in from scratch. However, the only changes to subroutine 6500 are to the PRINT statements at 6520, 6545, 6600, 6610, 6690 and 6700 and so it is worth editing the old version.

The assembler's new facilities mean that you can type in a program, list it, insert and append new lines to it and delete incorrect lines from it. You can also save and load programs to tape. As assembly language errors are very often destructive it is a good idea always to save a program before trying it out. Other features should be self-explanatory when you come to use the assembler.

Printing hex numbers

The machine code subroutine @PRINT at $800C will print characters on the Dragon's text screen. The character printed is set by storing its ASCII code in the A register before transferring control to the subroutine. It is often useful, especially when debugging a program, to be able to print the contents of the A or B register in hexadecimal form on the Dragon's screen. The problem is to convert the simple binary number in the B register into two ASCII characters in the sets 0-9 and A-F. Thus, each of the ASCII characters is determined by 4 bits, the first character to be printed by bits b7-b4 and the second by b3-b0. What we have to do is to write a program to separate the two groups of 4 bits and then convert them to the correct hex character.

The problem of separating the two groups of 4 bits is easy to solve using the AND operation and shifting. What about converting a simple 4 bit binary number to the correct ASCII character? Fortunately the ASCII codes from 0-9 and A-F run in sequence but there is a 'gap' between 9 and A. To see what this means examine the following table

binary	hex	ASCII code
0000	0	00110000
0001	1	00110001
0010	2	00110010
0011	3	00110011
0100	4	00110100
0101	5	00110101
0110	6	00110110
0111	7	00110111
1000	8	00111000
1001	9	00111001
1010	A	01000001
1011	B	01000010
1100	C	01000011
1101	D	01000100
1110	E	01000101
1111	F	01000110

For a value in the range 0 to 9 you can convert the 4 bit binary to its corresponding ASCII code by adding 00110000 or $30. If you examine the table for values greater than 9 you should be able to see that the 4 bit value can be converted to the corresponding ASCII code by adding $37 (this is simply the difference between the two codes).

Armed with this information the program is now fairly easy to write-

```
@PRINT  EQU     $800C
@HEX    STB     @TEMP
        LDA     @TEMP
        LSRA
        LSRA
        LSRA
```

```
        LSRA
        BSR     @HPRN
        LDA     @TEMP
        ANDA    #$0F
        BSR     @HPRN
        RTS

@HPRN   CMPA    #9
        BHI     @HP1
        ADDA    #$30
        JSR     @PRINT
        RTS
@HP1    ADDA    #$37
        JSR     @PRINT
        RTS
@TEMP   FCB     0
```

The first part of the program, starting at @HEX performs four left shifts to move the top for bits b7-b4 down into b3-b0. Notice that, by using LSRA, zeros are shifted into the top four bits. After this subroutine @HPRN is called to print the ASCII character corresponding to these four bits. The bottom 4 bits are then isolated by the ANDA #$0F which sets the top four bits to 0 leaving the rest unaltered (see bit manipulation - Chapter Five). Then subroutine @HPRN is called again to print the second four bits in hex. The only thing left to describe is the action of subroutine @HPRN. This first tests to see if the value in the A register is greater than 9. If it isn't it adds $30 and uses subroutine @PRINT to print the ASCII character on the screen. If it is bigger than 9 then $37 is added and @PRINT is used to print the ASCII character on the screen.

Even in this simple program there are a number of subtle points. Notice the way that the task of adding the correct value to the four bits is implemented as a subroutine so that it can be used more than once. The principle of using subroutines as often as possible applies just as much to assembly language programming as to BASIC. Also notice the way that the jump to the 'internal' subroutine @HPRN is achieved using a BSR and to the 'external' subroutine using JSR. This is such good 6809 assembly language practice that it is worth getting used to early on but exactly why it is such a good idea will have to wait

until Chapter Eleven. The branch instruction BHI is used because the four-bit numbers might as well be considered to be simple binary although, with the limited range of 0 to 15, two's complement could just as easily be used.

Although this hex number printing subroutine works it is worth adding a small program to test it. The following short program will print all the binary numbers from 0 to 255 in hex, over and over again.

```
@START  CLRB
@LOOP   BSR     @HEX
        LDA     #$20
        JSR     @PRINT
        INCB
        BRA     @LOOP
```

This should be typed in before the @HEX subroutine and EXECuted. You will see the screen fill with pairs of hex digits faster than you can read! The JSR @PRINT in the main program print a space (ASCII $20) between each pair of hex digits.

Multiple precision arithmetic - ADC and SBC.

So far, assembly language arithmetic has been limited to eight-bit arithmetic using the A and B registers and 16-bit arithmetic using the D register. Although these two ranges suffice for many applications it is important to know how to go about extending the range to anything that is desired. In this section, the problem that is tackled is the addition and subtraction of numbers that need four memory locations to represent them. Even though four memory locations, or 32 bits, can represent numbers in the range

0 to 4,294,967,296

using simple binary, and roughly

-2,147,483,648 to +2,147,483,647

using two's complement, it is possible that you might want to use even more memory locations to increase the range. This is quite easy once you have seen the general principle behind extended arithmetic. In practice it is this rather more complicated subroutine that is of practical use but it will have to wait for the next chapter to be completed,

Consider first the simpler problem of adding together a pair of 16-bit numbers but without using the D register. If the two numbers are stored in memory as shown -

	Most significant	Least significant
1st	@HNUM1	@LNUM1
2nd	@HNUM2	@LNUM2

then the first thing to do is add the two least significant eight bits together by -

```
LDA  @LNUM1
ADDA @LNUM2
```

This produces the correct answer for the least significant byte of the answer but, if the values stored in @LNUM1 and @LNUM2 are large enough, the correct answer may be so large that it needs nine bits to represent it. If you look back to the description of the C or Carry bit in Chapter Seven you will see that it is used to store the 'overflow' from any arithmetic. After adding together the least significant bytes and storing the result, the next stage is to add the most significant bytes of the pair of numbers. The only difficulty is that any 'carry' produced from the adding the least significant bytes should also be added into the result. The 6809 has two extra arithmetic operations that will perform addition or subtraction, taking into account any carry from a previous operation - ADC (ADd with Carry) and SBC (SuBtract with Carry). Both ADC and SBC work in roughly the same way as the familiar ADD and SUB instruction in that they leave the result of the operation in the register. The only real difference is that ADC and SBC take account of any carry from a previous operation and there is no ADCD nor SBCD operation. To add the two instructions to the BASIC assembler include the following four DATA statements -

```
REM BASIC ASSEMBLER V8.2

77 DATA ADCA,&H89,&H99,&HA9,&HB9,-1
78 DATA ADCB,&HC9,&HD9,&HE9,&HF9,-1
79 DATA SBCA,&H82,&H92,&HA2,&HB2,-1
80 DATA SBCB,&HC2,&HD2,&HE2,&HF2,-1
```

The final program for adding the pair of 16-bit numbers together is

```
LDA     @LNUM1
ADDA    @LNUM2
STA     @LANS
LDA     @HNUM1
ADCA    @HNUM2
STA     @HANS
```

which stores the most significant byte of the answer in @HANS and the least significant byte in @LANS.

The program for adding numbers taking four memory locations is just as simple if rather longer,

```
LDA     @1NUM1
ADDA    @1NUM2
STA     @1ANS
LDA     @2NUM1
ADCA    @2NUM2
STA     @2ANS
LDA     @3NUM1
ADCA    @3NUM2
STA     @3ANS
LDA     @4NUM1
ADCA    @4NUM2
STA     @4ANS
```

where the two numbers are stored in memory locations @1NUM1 to @4NUM1 and @1NUM2 to @4NUM2 and the result is stored in @1ANS to @4ANS. To subtract numbers taking four memory locations just use the same program given above but replace the ADDA by SUBA and ADCA by

SBCA. Notice that the first add or subtract in each of these programs ignores the carry bit.

It is possible to write a general program that will add and subtract any number of memory locations but this requires yet a little more assembly language and it is covered in the next chapter. The problem is that to operate on a variable number of memory locations requires some way of keeping track of the all the addresses. The idea of using labels such as @1NUM1 to @1NUM4 is all very well for four memory locations, but imagine trying to use the same method for 10 or 20 memory locations!

Example: a 'squash' game

The program that is the subject of this section plays the familiar video game of squash. It is far from a finished game in the sense that it doesn't include scoring or any sort of 'user friendliness'. However, it is a complete program from the point of view of the essential working parts of the game. The ball, a square block, bounces around the screen. A small bat can be moved to the right and left using the left and right arrow keys and the ball will bounce off it if it happens to be in the correct position at the correct time. The most important thing about the program, however, is that it is an excellent example of assembly language arithmetic in action and of the way in which a larger assembly language program is put together.

Rather than tackle the problem in one go it is easier to approach the finished program in three main stages -

first, bounce a solid block around the screen;
second, add the bat and the logic to move it;
third, add the logic which will bounce the ball off the bat.

Bouncing a 'ball'

The techniques involved in bouncing a ball around the Dragon's text screen should be well known to you from BASIC. (For general information about the principles involved consult Chapter Six of "The Complete Programmer" published by Granada.) The idea is to keep a record of the ball's position in two variables X and Y. These are updated by adding the current values of two other variables XV and YV i.e.

```
Xnew = Xold + XV
Ynew = Yold + YV
```

The values XV and YV can be thought of as velocities governing the motion in the X and Y direction. To make the ball appear to move, it has to be displayed at the screen location given by X,Y, then removed by printing a blank at the same location and the co-ordinates updated and then the whole cycle if repeated. To make the ball appear to bounce all that is necessary is to reverse one of the velocities when the ball 'hits' a wall. For example, if the ball hits a horizontal wall the vertical velocity has to be reversed, i.e. VY = -VY, and likewise the horizontal velocity has to be reversed for a vertical wall.

Perhaps the best way to illustrate the methods employed in bouncing a ball around the screen is to give a BASIC program. This not only shows how things work, it provides a model for the assembly language program and will give you a good idea of the speed advantage inherent in assembler.

```
20 YCORD=0
30 XVEL=1
40 YVEL=1
50 C$=CHR$(128)
60 GOSUB 1000
70 C$=" "
80 GOSUB 1000
90 GOSUB2000
100 GOTO50


1000 LOC=XCORD+32*YCORD
10 10 PRINT@LOC,C$;
1020RETURN


2000 XCORD=XCORD+XVEL
2010 IFXCORD=0ORXCORD=31THENXVEL=-XVEL
2020 YCORD=YCORD+YVEL
2030 IFYCORD=0ORYCORD=15THENYVEL=-YVEL
2040 RETURN
```

Subroutine 1000 prints the character stored in C$. Subroutine 2000 updates the X and Y values and checks for a possible 'collision' with the edge of the screen! If a collision is detected then the appropriate velocity is reversed. Notice that the velocities are either 1 or -1 for simplicity.

Implementing the above BASIC program in assembler is mainly a matter of writing a subroutine to 'print' the ball on the screen and to update the co-ordinates and velocities. The first problem to tackle is to write a subroutine that will print any character on the screen at XCORD,YCORD. This can be done by calculating the screen memory location corresponding to the position XCORD,YCORD and then storing the appropriate character code in it. Calculating the text screen memory location is easy, in theory at least, using -

address = $400 + XCORD + 32*YCORD

(For the details of this equation see "The Anatomy of the Dragon", also published by Sigma.) Before this expression can be implemented in assembler it is necessary to decide on a suitable representation for XCORD and YCORD. The possible range for XCORD is 0 to 31 and for YCORD 0 to 15. Both of these values can easily be stored in a single memory location using simple binary as there is no possibility of a negative number ever occurring. However, as the address calculated by the above expression is a 16-bit quantity we must be prepared to combine eight-bit values and 16-bit arithmetic (see Chapter Six). Also, as the velocities are going to have to vary between 1 and -1, two's complement arithmetic is going to enter into the program somewhere. So, assuming that @XCORD and @YCORD label single memory locations used to hold the ball's co-ordinates and @CHAR labels a single memory location holding the character code character to be 'printed', the subroutine to calculate the screen memory location is -

```
@DRAW   LDB     @YCORD
        CLRA
        ASLB
        ASLB
        ASLB
        ASLB
        ASLB
        ROLA
        ADDB
```

```
@XCORD  ADCA    #$00
        ADDD    #$0400
```

The first part of the program multiplies the value in @YCORD by 32. This is achieved using five shift lefts. The calculation starts off using eight-bit arithmetic but, as the final answer is going to be 16 bits, the A register is cleared to produce a valid 16-bit number in the D register. Since the value lies in the range 0 to 15, i.e. it is held in the first four bits, there is no chance of an overflow for the first four shifts but the fifth shift could produce a number outside the range of eight-bit simple binary representation. If you look back at the definition of ASL you will see that b7 is shifted into the C bit and so, if the fifth shift causes an overflow it will be in the C bit. The question is how to move this C bit into the A register. This can be achieved using the ROLA instruction which shifts the C bit into b0 of the A register which is just what we want. So multiplication by 32 can be achieved by five ASLB instructions followed by one ROLA, leaving a valid simple binary number in the D register. The next part of the program adds the @XCORD value. As this is an eight-bit value it has to be added to the D register in a round about way. First, the least significant bytes are added using ADDB @XCORD. This may produce a carry which then has to be added to the most significant byte in the A register. This is done using an ADCA #$00 instruction. To understand this just ask yourself what the eight most significant bits of @XCORD are - the answer is eight zeros. Adding zero to take account of the carry is something that is often encountered in eight-bit/16-bit register arithmetic. Now that @XCORD has been added, the only operation left is the addition of $400 to give the final address. As this is a constant there is no trouble about representing it as a 16-bit simple binary value and adding it to the D register using ADDD #$0400.

The reason why this piece of arithmetic was so complicated is that the values involved in the calculation were mainly best represented as eight-bit quantities but the answer has to be a 16-bit quantity. This mixing of the number of bits couldn't have been completely avoided even by representing all the values as 16-bit values - there is no 16-bit shift - and the alternative way of multiplying by 32 using the MUL instruction uses eight-bit values in the A and B registers. In most assembly language programming there is no avoiding this sort of 'mixed precision arithmetic'.

At the end of this routine we have in the D register the address of the screen memory location in which the character code has to be stored. The question now is what to do with it? So far we have no way of storing a value in a

memory location whose address has been worked out in the course of a program. What we need is the facility to say 'use the number stored in a register as the address of a memory location'. The 6809 does provided this facility as a separate addressing mode called 'indexed addressing'. Indexed addressing is in fact a whole family of addressing modes and for this reason it is given a whole chapter - the next one - to itself! However, one form of indexed addressing is so useful that its introduction cannot be postponed any longer.

The 6809 has another register called the X register that is a full 16 bits in length and can be used to hold an address. It can be loaded and stored using the instructions LDX and STX in much the same way that the D register can. However, it is different from the D register in that it isn't used to do calculations. It is primarily an 'addressing' or 'pointer' register. That is, it is used to hold the address of the memory location that another instruction will operate on. For example, the instruction -

```
STA ,X
```

will store the contents of the A register in the memory location whose address is stored in the X register. Don't worry for the moment about the "," in front of the X. All it is used for at this stage is to alert the assembler to the fact that indexed addressing is being used.

Now we have the solution to what to do with the address of the screen location left in the D register -

```
STD     @SCREEN
LDX     @SCREEN
LDA     @CHAR
STA     ,X
RTS
```

The first two instructions transfer the contents of the D register to the X register by way of the pair of memory locations labelled by @SCREEN (a more direct way to do this will be introduced in the next chapter). The next instruction loads the A register with the character code and the STA ,X stores this value into the screen memory location whose address is in X. Finally RTS completes the @DRAW subroutine.

The next subroutine that is required it the subroutine that updates the co-ordinates and changes the velocities when there is a collision. The @UPDATE subroutine is another exercise in assembly language arithmetic but this time with the added interest of using conditional branches to test the results. The subroutine is best thought of in two parts -

update @XCORD and check for a bounce
update @YCORD and check for a bounce

Updating @XCORD is just a matter of adding the contents of @XVEL to it - the only complication is that @XVEL has to be a two's complement number to represent 1 and -1. In practice this causes no problem because the values of the co-ordinates are small and positive and so they can be just as easily considered to be in two's complement representation as in simple binary. (Notice that simple binary numbers in the range 0 to 127 have exactly the same bit pattern in two's complement.) This settled, the subroutine is easy to write by using the BASIC subroutine as a guide -

```
@UPDATE  LDA     @XCORD
         ADDA    @XVEL
         STA     @XCORD
         BNE     @SKIP1
         NEG     @XVEL
@SKIP1   CMPA    #31
         BNE     @SKIP2
         NEG     @XVEL
@SKIP2   LDA     @YCORD
         ADDA    @YVEL
         STA     @YCORD
         BNE     @SKIP3
         NEG     @YVEL
@SKIP3   CMPA    #15
         BNE     @SKIP4
         NEG     @YVEL
@SKIP4   RTS
```

The section of the program from @UPDATE to @SKIP2 updates @XCORD and @XVEL and the section from @SKIP2 to @SKIP4 does the same for

@YCORD and @YVEL. As these two halves are so similar, only the first half will be described. All of the arithmetic in this subroutine can be carried out in eight bits using the A register. The contents of @XCORD are added to @XVEL and then stored back into @XCORD. The only thing to notice here is that @XVEL could be negative (i.e. -1) and so @XCORD could decrease in value. The BNE checks for a zero result. If one is found the @XVEL has its sign changed using the NEG instruction. Notice that there is no need to use CMPA #0 because the condition code bits are set by the STA instruction. To check for the other extreme value, however, it is necessary to use CMP #31. Once again, however, a BNE instruction is used to skip round the NEG @XVEL instruction. The second half of the program updates @YCORD and @YVEL in roughly the same way.

All that now remains is to write the main program that uses the subroutines-

```
@START   BSR     @UPDATE
         LDA     #$80
         STA     @CHAR
         BSR     @DRAW
         LDA     #$8F
         STA     @CHAR
         BSR     @DRAW
         BRA     @START
@XCORD   FCB     0
@YCORD   FCB     0
@XVEL    FCB     1
@YVE     FCB     1
@CHAR    FCB     0
@SCREEN  FDB     0
```

This works in the same way that the BASIC main program given earlier did. The program is in the form of a loop and each time through the @UPDATE subroutine is used to move the ball and the @DRAW subroutine is used twice, once to display the ball and once to print a blank to remove it. If you were to put the main program together with the subroutines and EXEC the program you would see the ball bounce around the screen very fast. The only trouble is that at the moment the BASIC assembler cannot handle the instructions LDX and STA ,X.

Adding simple indexed addressing to the BASIC assembler

The simplest way to add indexing to the BASIC assembler is to test for a "," in the address field and then set TYPE=3 as a result. There are a few other matters to be cleared up but the changes are not at all difficult-

```
1 REM BASIC ASSEMBLER V8.3

81 DATA LDX,&H8E,&H9E,&HAE,&HBE,-1

5035 IF TYPE=3 THEN RETURN

5526 IF L$="," THEN GOTO 5800

5800 TYPE=3
5810 AF$="&H84"
5820 RETURN

6260 IF TYPE=2 OR TYPE=1 OR TYPE=3 THEN POKE P,A
```

Subroutine 5800 will eventually be expanded to cope with the job of coding other varieties of indexed addressing.

Testing and perfecting the bounce program.

Now that the BASIC assembler can handle the indexed addressing mode used in @DRAW the whole program can be typed in and tested. Remember to include all of the two subroutines following the main program. Once you have entered the program correctly you can EXECute it and see the results. You might be a little disappointed at what you see as the ball zips about the screen a little too fast! In fact it zips about the screen so fast that it is difficult to see the whole ball at any given moment. The solution to this problem is to include a delay subroutine between the first and second uses of the @DRAW

subroutine. The standard way of implementing any sort of delay in BASIC or in assembler is via a time-wasting loop - a 'delay loop'.

```
@DELAY  LDD     @TIME
@DLOOP  SUBD    #1
        BNE     @DLOOP
        RTS
@TIME   FDB     $1000
```

The longest possible delay is produced by a value of @TIME equal to 0. If you insert BSR @DELAY following the first BSR @DRAW and re-run the program you will see a distinct improvement. Try altering the value stored in @TIME to see the effect of different delays.

Adding the bat

Producing a moving bat controlled by two keys on the keyboard is in some ways easier than moving and bouncing the ball and in some ways more difficult. Rather than being a single character like the ball, the bat has to be made up from a number of characters to be large enough to make the game playable. However, the actual positioning of the bat is easier because it only moves from side to side.

The bat is made up from three solid blocks and its position on the screen is controlled using the x co-ordinate of the leftmost block (see fig 8.1). The problem of drawing the bat is considerably eased by the fact that it only moves horizontally. If the x co-ordinate of the lefthand block of the bat is stored in @XBAT, then the address of the corresponding screen memory location is -

address = base + @XBAT

where 'base' is the address of the screen memory location on the far left of the same line that the bat is on. If the bat is to be printed on line 14, base is equal to $05C0. The subroutine to draw the bat at @XBAT is -

```
@DBAT   LDB     @XBAT
        CLRA
```

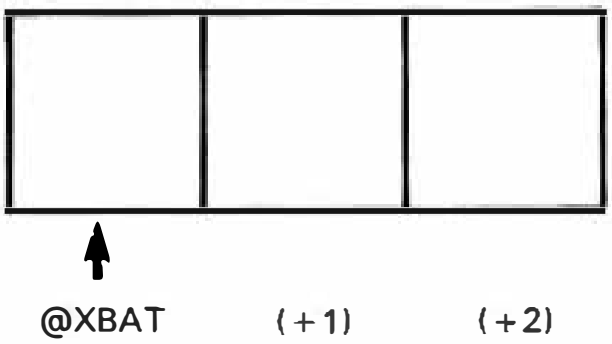


Fig 8.1 The bat

```
ADDD #$05C0
$05C0
STD         @ADDRESS
LDX         @ADDRESS
LDA         @CHAR
STA         ,X
LDD         @ADDRESS
ADDD        #1
STD         @ADDRESS
LDX         @ADDRESS
LDA         @CHAR
STA         ,X
LDD         @ADDRESS
ADDD        #1
STD         @ADDRESS
LDX         @ADDRESS
LDA         @CHAR
STA         ,X
RTS
```

The way that this program works is straightforward, in that it first calculates the address of the screen location corresponding to the left hand block of the bat and then stores the character code in @CHAR in it, but as the bat consists of three blocks, it then has to store the same character code at @ADDRESS+1 and @ADDRESS+2. This it achieves by using the D register to add one to the contents of @ADDRESS and then using it in the X register to govern where the contents of the A register are stored. This is a rather long way of storing something in three consecutive memory locations. There is a more concise way of achieving the same ends and this will be described in the next chapter.

All that is needed now is a second subroutine to update the bat's position depending on which of the two arrow keys is pressed. The update procedure is simple. If the left arrow key is pressed then @XBAT should be decreased by one and if the right arrow key is pressed @XBAT should be increased by one. There are a number of extra considerations apart from just moving the bat that make the update subroutine just a little more complicated than this. In the first place it is important to make sure that the bat doesn't go off the edges of the screen and in the second it would be easier to play the game if the keyboard had a fast auto-repeat feature. The first requirement merely involves checking that the value in @XBAT is greater than 0 before subtracting one from it and less than 29 before adding one to it. The auto-repeat facility is also easy to produce once you know that the Dragon maintains a table of keys pressed at $150 to $159 and the way to fool it into thinking that a new key has been pressed is to store $FF in each of these locations. In fact, as a press of either arrow key is recorded in memory locations $151, $157 and $158 these are all that have to be set to $FF in this case. The update subroutine is -

```
@UPBAT  JSR     @KEYB
        BEQ     @REPKEY
        CMPA    #$08
        BNE     @RARR
        LDB     @XBAT
        BEQ     @REPKEY
        DEC     @XBAT
@RARR   CMPA    #$09
        BNE     @REPKEY
        LDB     @XBAT
        CMPB    #29
        BEQ     @REPKEY
```

```
        INC     @XBAT
@REPKEYLDA      #$FF
        STA     @ROLL
        STA     @ROLL2
        STA     @ROLL3
        RTS
@ROLL   EQU     $151
@ROLL2  EQU     $157
@ROLL3  EQU     $158
@KEYB   EQU     $8006
```

The first instruction uses the machine code subroutine @KEYB to read the keyboard. This subroutine returns the ASCII code of any key pressed in the A register or returns zero if no key is pressed. If no key is pressed the second instruction transfers control to the end of the @UPBAT subroutine. The rest of the subroutine checks for a left or right arrow key press and only updates @XBAT if it is in the correct range. The last part of the subroutine, starting at @REPKEY implements the auto-repeat key described earlier.

All that is now required is a new version of the main program given earlier.

```
@START  BSR     @UPDATE
        LBSR    @UPBAT
        LDA     #$80
        STA     @CHAR
        BSR     @DRAW
        BSR     @DBAT
        BSR     @DELAY
        LDA     #$8F
        STA     @CHAR
        BSR     @DRAW
        BSR     @DBAT
        BRA     @START
@XCORD  FCB     0
@YCORD  FCB     0
@XVEL   FCB     1
@YVEL   FCB     1
@CHAR   FCB     0
@ADD
RESS    FDB     0

@XBAT   FCB     12
```

(Notice the need to use LBSR to jump to the @UPBAT subroutine because it is now too far away from the start of the program to be reached using BSR). If you type this program in along with all the subroutines used and then EXE-Cute the program you will see the ball bounce around the screen as before but now there is a bat, initially positioned roughly in the middle of the screen, that you can move from one side of the screen to the other using the arrow keys. The only trouble is that the ball passes right through the bat just as if it wasn't there!

Detecting the bat

This is the final part of the squash program, consisting of a single new subroutine to detect the ball hitting the bat and to act upon it. There are a number of ways that the ball colliding with the bat could be detected but the most instructive from the point of view of assembler is by comparing the

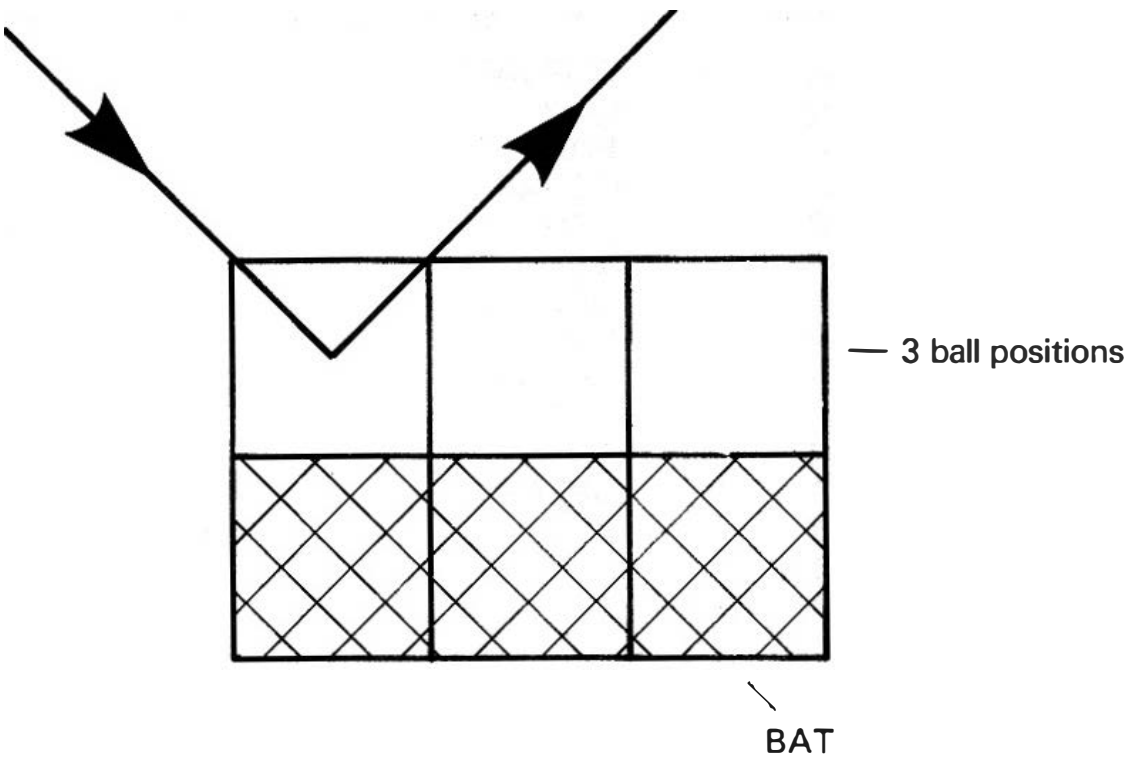


Fig 8.2 3 possible ball positions "hitting" the bat

co-ordinates. If the ball is in contact with the bat it can be in any one of the threepositionsshown in fig 8.2. This implies that you can detect when the ball should be bounced from the bat by the following condition -

@YCORD = bat's horizontal position - 1

and

0 < = XCORD - XBAT < = 2

To express this another way, to detect a bounce the subroutine has to test that the difference between @XCORD and @XBAT lies in the range 0 to 2:

```
@BBOUNCE  LDA    @YCORD
          CMPA   #$13
          BNE    @NBOUNCE
          LDA    @XCORD
          SUBA   @XBAT
          BMI    @NBOUNCE
          CMPA   #2
          BGT    @NBOUNCE
          NEG    @YVEL
@NBOUNCE  RTS
```

If the @YCORD of the ball is just one line above the bat (as shown in fig 8.2) then the x co-ordinates are checked. This is done by subtracting them. If the result is negative then the difference is less than zero so a bounce cannot occur. If the difference if positive it is compared to 2, once again if it is greater a bounce cannot occur. If the co-ordinates do satisfy all the conditions the the bounce is implemented simply by reversing @YVEL.

The @BBOUNCE subroutine is easily added to the previous main program by inserting a LBSR @BBOUNCE following the LBSR @UPBAT instruction. When you EXEC the entire program you will see the ball bouncing round as before but now it will bounce off the bat if you happen to get it into the correct position! Obviously, to make a finished game out of the program you would have to add extra features, including a count of the hits and misses, but you should be able to do this on your own.

The complete program

To make sure that you get everything in the right place the output of the BASIC assembler for the squash program is given below -

```
7000    8D    26     @START BSR @UPDATE
7002    17    AC     LBSR @UPBAT
7005    17    D4     LBSR @BBOUNCE
7008    86    80     LDA #$80
700A    B7    7022   STA @CHAR
700D    8D    44     BSR @DRAW
700F    8D    6B     BSR @DBAT
7011    8D    5E     BSR @DELAY
7013    86    8F     LDA #$8F
7015    B7    7022   STA @CHAR
7018    8D    39     BSR @DRAW
701A    8D    60     BSR @DBAT
701C    20    E2     BRA @START

701E    0            @XCORD FCB 0
701F    0            @YCORD FCB 0
7020    1            @XVEL FCB 1
7021    1            @YVEL FCB 1
7022    0            @CHAR FCB 0
7023    0     0      @SCREEN FDB 0
7025    0     0      @ADDRESS FDB 0
7027    C            @XBAT FCB 12
7028    B6    701E   @UPDATE LDA @XCORD
702B    BB    7020   adda @XVEL
702e    B7    701E   STA @XCORD
7031    26    3      BNE @SKIP1
7033    70    7020   NEG @XVEL
7036    81    1F     @SKIP1 CMPA #31
7038    26    3      BNE @SKIP2
703A    70    7020   NEG @XVEL
703D    B6    701F   @SKIP2 LDA @YCORD
7040    BB    7021   ADDA @YVEL
7043    B7    701F   STA @YCORD
7046    26    3      BNE @SKIP3
```

```
7048    70    7021    NEG @YVEL
704B    81    F       @SKIP3 CMPA #15
704D    26    3       BNE @SKIP4
704F    70    7021    NEG @YVEL
7052    39            @SKIP4 RTS

7053    F6    701F    @DRAW LDB @YCORD
7056    4F            CLRA
7057    58            ASLB
7058    58            ASLB
7059    58            ASLB
705A    58            ASLB
705B    58            ASLB
705C    49            ROLA
705D    FB    701E    ADDB @XCORD
7060    89    0       ADCA #$00
7062    C3    400     ADDD #$0400
7065    FD    7023    STD @SCREEN
7068    BE    7023    LDX @SCREEN
706B    B6    7022    LDA @CHAR
706E    A7    84      STA ,X
7070    39            RTS

7071    FC    707A    @DELAY LDD @TIME
7042    83    1       @DLOO SUBD #1
7077    26    FB      BNE @DLOOP
7079    39            RTS
707A    10    0       @TIME FDB $1000
707C    F6    7027    @DBAT LDB @XBAT
707F    4F            CLRA
7080    C3    5C0     ADDD #$05C0
7083    FD    7025    STD @ADDRESS
7086    BE    7025    LDX @ADDRESS
7089    B6    7022    LDA @CHAR
708C    A7    84      STA ,X
708E    FC    7025    LDD @ADDRESS
7091    C3    1       ADDD #1
7094    FD    7025    STD @ADDRESS
7097    BE    7025    LDX @ADDRESS
```

```
709A    B6    7022    LDA @CHAR
709D    A7    84      STA ,X
709F    FC    7025    LDD @ADDRESS
70A2    C3    1       ADDD #1
70A5    FD    7025    STD @ADDRESS
70A8    BE    7025    LDX @ADDRESS
70AB    B6    7022    LDA @CHAR
70AE    A7    84      STA ,X
70B0    39            RTS


70B1    BD    8006    @UPBAT JSR @KEYB
70B4    27    1A      BEQ @REPKEY
70B6    81    8       CMPA #$08
70B8    26    8       BNE @RARR
70BA    F6    7027    LDB @XBAT
70BD    27    11      BEQ @REPKEY
70BF    7A    7027    DEC @XBAT
70C2    81    9       @RARR CMPA #$09
70C4    26    A       BNE @REPKEY
70C6    F6    7027    LDB @XBAT
70C9    C1    1D      CMPB #29
70CB    27    3       BEQ @REPKEY
70CD    7C    7027    INC @XBAT
70D0    86    FF      @REPKEY LDA #$FF
70D2    B7    151     STA @ROLL
70D5    B7    157     STA @ROLL2
70D8    B7    158     STA @ROLL3
70DB    39            RTS
                      @ROLL EQU $151
                      @ROLL2 EQU $157
                      @ROLL3 EQU $158
                      @KEYB EQU $8006


70DC    B6    701F    @BBOUNCE LDA @YCORD
70DF    81    D       CMPA #13
70E1    26    F       BNE @NBOUNCE
70E3    B6    701E    LDA @XCORD
70E6    B0    7027    SUBA @XBAT
70E9    2B    7       BMI @NBOUNCE
```

```
70EB    81    2       CMPA #2
70ED    2E    3       BGT @NBOUNCE
70EF    70    7021    NEG @YVEL
70F2    39            @NBOUNCE RTS
```

Conclusion

In this chapter you have seen the way that a large assembly language program is built up from subroutines. You should now have the flavour of assembly language programming and if you have typed in and tried the squash program an idea of some of the difficulties and rewards. The BASIC assembler is working at its limits to assemble a program of over 100 lines and fortunately the examples in the rest of the book do not approach this length! It should now be clear that if you are going to use assembler at all often you should purchase a good, fast assembler. Study the squash program until you understand what each part is doing. To check that you do understand try taking it a little further, add a routine to make the ball go out of play if you miss it, a score line and so on.

Chapter Nine

The addressing registers- indexed addressing

It may come as something of a shock to discover that there are still five registers inside the 6809 yet to be discussed! All of these registers are concerned with manipulating addresses rather than data and so it makes sense to treat them all together in one chapter. The idea of using a register to specify an address was briefly introduced in the previous chapter in connection with the the squash program. Even though this was a fairly simple example it was difficult to avoid using a register to specify a memory location whose address had been calculated in the course of the program. This idea of calculating the address of a memory location is a very general and powerful and forms part of almost all assembly language programs of any size. As a 6809 address uses 16 bits it is clear that a general addressing register has to be able to store 16 bits. What is less obvious is that the range of operations that are generally applied of addresses is smaller than the range applied to data. For this reason, the6809's addressing registers are associated with a different set of operations from the A and B registers. The use of the addressing registers and their associated operations gives the 6809 a whole family of extra addressing modes collectively known as 'indexed addressing'.

Indexed addressing is such a large subject that it would be possible to write a reasonable length book on how the indexed addressing modes work and how they are used! Fortunately, once you have picked up the idea of calculating an address using the addressing registers, then all the different variations are fairly easy to understand. What is more difficult is trying to see what use some of the more exotic indexed addressing modes might be useful for. The best answer to this question is that when you are writing a program that needs or would benefit from their use you will soon recognise it! However, to help you get started with indexed addressing there are some 'sound' examples at the end of the chapter!

The addressing or pointer registers

There are five registers within the 6809 that are exclusively concerned with the matter of specifying addresses. Four of them are full addressing registers, in the sense that they are 16 bits long and able to hold a complete address, and one of them, the DP or 'Direct Page' register, is only eight bits long and has to be used in conjunction with other information to determine a full address. In fact the DP register is so different and special that it is better to treat it in a section all on its own towards the end of this chapter. The four full addressing or 'pointer registers' are called the X, Y, U and S registers. Although all four of them can be used in the same way as far as index addressing goes, the X and Y registers are a little more limited than the U and S registers. In particular, the U and S registers are involved in so-called 'stack operations' which are explained in the next chapter. Also the S register is used by the 6809 to keep track of return addresses during subroutine calls and returns. For this reason the X and Y registers are called the 'index registers' and the U and S registers are called the 'stack pointers'. Despite the different names used for the registers, it is important to realise that as far as indexed addressing is concerned the four registers are identical. However, even though they are identical in theory, the Dragon uses the S register extensively and it is safer to use the index registers for indexing and the stack registers mainly for stack operations, as described in the next chapter.

Operations on the pointer registers

Although the pointer registers are used for a different purpose to the A and B registers there are a number of operations that they share with them. For example, in the same way that you can load and store the A and B registers using LDA, LDB, STA and STB you can load and store the pointer registers using similar instructions e.g. LDX loads the X register. Because they are so similar to the already familiar A and B register operations, rather than treat each of the operations on the pointer registers at length, it should be sufficient to list them:

the load instructions

```
LDS     address
LDU     address
LDX     address
LDY     address
```

the store instructions

```
STS     address
STU     address
STX     address
STY     address
```

the compare instructions

```
CMPS    address
CMPU    address
CMPX    address
CMPY    address
```

The action of each of these instructions should be obvious. However, it is worth pointing out that each pointer register is 16 bits long. For example, LDX address loads the X register taking the most significant byte from 'address' and the least significant byte from 'address + 1'. Apart from these three forms of instruction, load, store and compare, there are some completely new ones that also work with the pointer registers and they will be described later in this chapter.

Simple indexing

The idea of using a register to hold the address to be used by an instruction is not a difficult one. Any instruction that can be used with indexed addressing can take its address from any of the four pointer registers. The way that simple indexing is indicated in assembler is by writing the name of the pointer register that contains the address in the address field of the instruction preceded by a comma. So for example,

```
LDA,X
```

loads the A register from the memory location whose address is stored in the X register. Similarly,

```
STA ,U
```

stores the contents of the A register in the memory location whose address is stored in the U register.

You may be wondering what the comma in front of the register's name is for. The answer is that you can write a number in the range -32768 to32767 in front of the comma that will automatically be added to the contents of the register and the result used as the address of the memory location. For example, if the X register contains $6000 the instruction

```
LDA 3,X
```

will load the A register from $6003. The number to the left of the comma is known as the 'offset' because, rather like the offset used in relative addressing, it indicates how far away, in terms of memory locations, the data is from the address currently in one of the pointer registers. So the general form of 'simple' or 'constant offset indexing' as it is more properly called is -

```
mnemonic offset,pointer register
```

Some valid examples of constant offset indexing are -

```
LDA     -32,X    (load the A register from X-32)
ADDB    5,U      (add the contents of U+5 to the B
                  register)
INC     1,Y      (add one to memory location Y+1)
```

Notice that the LDA ,X is a shorthand way of writing LDA 0,X.

It is important to realise the contents of the pointer isn't altered in any way by being used in indexed addressing. For example, LDA 1,X only adds 1 to X to work out the address to be used. The contents of X after the instruction are the same as they were before the instruction. In other words, the address calculation only produces a temporary result.

As an example of how convenient constant offset indexing can be, consider subroutine @DBAT in the previous chapter. The problem that the subroutine solved was to calculate an address and then store the contents of @CHAR in 'address', 'address'+1 and 'address'+2. Now this can be solved much more directly using constant offset indexing

```
@DBAT   LDB     @XBAT
        CLRA
        ADDD    #$05C0
        STD     @ADDRESS
        LDX     @ADDRESS
        LDA     @CHAR
        STA     0,X
        STA     1,X
        STA     2,X
        RTS
```

The address is calculated in the same way and then transferred to the X register using @ADDRESS as an intermediary. Then the A register is loaded from @CHAR and stored in the three consecutive memory locations given by X, X+1 and X+2. If you compare both versions of the subroutine you will immediately see that indexed addressing not only makes the program shorter, it also makes it easier to understand. The subroutine would be even shorter and easier to understand if there was some way of transferring the contents of the D register directly into the X register. There is indeed an instruction that will transfer the contents of any register to any other.

The TFR and EXG instructions

Although the two instructions TFR (TransFeR registers) and EXG (ÉXchanGe registers) are not really connected with the main subject of this chapter, they can now be described because all of the 6809's registers have been introduced, if not fully discussed. The instruction -

```
TFR r1,r2
```

will transfer the contents of register r1 into register r2 without altering the contents of r1. For example,

```
TFR A,B
```

results in the contents of the A register also being stored in the B register. (In other words, both registers now contain the same value.) The instruction -

```
EXG r1,r2
```

is superficially like the TFR instruction but it transfers the contents of register r1 into register r2 AND it also transfers the original contents of r2 into r1. That is, it swaps the contents of the registers. For example,

```
EXG A,B
```

exchanges the contents of the A and B registers. After the instruction, the A register contains the value that was in the B register and the B register contains the value that was in the A register.

The instructions TFR and EXG can be used to transfer or exchange the contents of any pair of 6809 registers that are the same size. That is, you can transfer or swap any pair of eight-bit registers, and any pair of 16-bit registers, but you cannot transfer or exchange between an eight- and 16-bit register. For example,

```
TFR A,CC
```

is valid because the both the A and CC (Condition Code register) are both eight-bit registers, but

```
TFR A,X
```

is not allowed because the A register is eight-bit and the X register is 16-bit. However, as the D register is 16-bit and is made up of the A and B registers this forms a link between the 6809's eight- and 16-bit registers that can be used to transfer and exchange values between all the registers.

Using the TFR instruction and constant offset indexed addressing, the @DBAT subroutine becomes -

```
@DBAT   LDB     @XBAT
```

```
CLRA
ADDD    #$05C0
TFR     D,X
LDA     @CHAR
STA     0,X
STA     1,X
STA     2,X
RTS
```

and now looks particularly simple!

Accumulator offset indexed addressing

Constant offset indexed addressing is very useful if you want to use a few memory locations next to each other, as in the @DBAT subroutine, but consider the problem of zeroing a whole section of memory. You could use something like -

```
LDX     #@START
CLR     0,X
CLR     1,X
CLR     2,X
CLR     3,X
```

and so on until the appropriate amount of memory had been cleared. (The address of the first memory location to be cleared is labelled by @START.) However, if the program had to clear say 100 locations this method would result in rather a long program.

What is required to solve the problem is the facility to specify a 'variable offset'. This is provided by the next level of indexed addressing, 'accumulator offset indexed addressing'. As you might be able to guess from its name, accumulator offset indexed addressing works in the same way as constant offset indexed addressing except that the offset is taken to be the current contents of one of the accumulators A, B or D. For example,

```
LDA B,Y
```

loads the B register from the memory location whose address is obtained by adding the contents of the B register to the contents of the Y register. Notice that once again neither the contents of the B register nor the contents of the Y register are affected by this instruction. The contents of the eight-bit registers A and B are treated as two's complement values so, that using these registers, you can address the current contents of a pointer register from -128 to +127. The value in the D register is also treated as a two's complement value when it is added to the pointer register so this gives a range of -32768 to +32767 from the current contents of the pointer register. You may be a little puzzled by an instruction like -

```
LDA A,Y
```

because the A register's value is changed by this instruction. There is no need to worry! The original contents of the A register are added to the contents of the Y register and then this result is used as the address of the memory location that A is loaded from.

Using accumulator offset indexed addressing the problem of clearing 100 memory locations is easily solved -

```
        CLRA
        LDX     #@START

@LOOP   CLR     A,X
        INCA
        CMPA    #99
        BLE     @LOOP
```

which can be further simplified if the memory can be cleared starting from @START+99 and working down to @START -

```
        LDA     #99
        LDX     #@START

@LOOP   CLR     A,X
        DECA
        BNE     @LOOP
```

If you want to clear (or do anything else!) to more memory locations than can be addressed by an eight-bit two's complement offset then you can always use the D register to specify a 16-bit two's complement offset. However, you should keep in mind the fact that while the D register is being used to specify an offset, both the A and B registers are unavailable for calculations unless the D registers value is saved and restored. One possible solution is to use the EXG instruction to swap the D register's value with one of the other pointer' register's.

Auto increment/auto decrement indexing

There is a second way of 'stepping' through a series of memory location using indexed addressing, known as 'auto increment/decrement indexing'. The basic idea is that you can specify that the pointer register automatically has either 1 or 2 added to or subtracted from it each time it is used. For example,

```
LDA ,X+
```

means "load the A register from the memory location whose address is stored in the X register and then add one to X". In general, writing one plus sign after the pointer register's name is taken to mean that 1 should be added to the register AFTER the operation is complete, thus leaving the register 'pointing' at the next memory location. Writing two plus signs after the register's name will cause it to be incremented by 2 following the operation. Unfortunately this is where it finishes. Writing three plus signs will simply give you an error message. This adding one or two to the pointer register after the operation is known as 'auto increment mode'. If you want to use memory locations that differ by 1 or 2 in ascending order then auto increment mode is by far the best method. For example, the program to clear 100 memory locations is best written as

```
        CLRA
        LDX     #@START
@LOOP   CLR     ,X+
        INCA
        CMPA    #99
        BLE     @LOOP
```

Notice that auto increment is only allowed with a constant offset of zero.

'Auto decrement' works in a very similar way to auto increment apart from subtracting 1 or 2 BEFORE the instruction is completed. For example,

```
LDA ,-X
```

will first subtract 1 from the contents of the X register and then use the result as the address of the memory location that A is loaded from. Notice that the minus sign is written in front of the pointer to indicate that the subtraction is done before the contents of the register are used as the address of the memory location. In the same way, writing two minus signs in front of the pointer register's name will subtract 2 from the register's value before it is used as an address. Notice that in both cases the addition or subtraction actually alters the value stored in the pointer register. Some valid example of auto increment and decrement are -

`LDA ,-X`	subtract 1 from the X register and then load the A register from the memory location that it 'points' at
`STA ,--U`	subtract 2 from the U register and then store the A register in the memory location that it 'points' at
`CLR ,Y+`	clear the memory location that Y 'points' at and then add one to Y
`NEG ,S++`	performs the NEG operation on the memory location whose address is stored in S and then adds 2 to S

In general, if you want to use a series of memory locations that differ by 1 or 2, use auto increment or auto decrement indexed addressing - it is fast and efficient. If you want to use memory locations that differ by more than 2 or that differ by a variable amount then use accumulator offset indexed addressing and use the arithmetic operations on the accumulators to update the address.

The effective address - the LEA instruction

All of the indexed addressing methods described so far have one thing in common, they all work out an address to be used by the instruction that they are part of. The address that is calculated in the course of indexed addressing is usually called the 'effective address'. On some occasions it would be an advantage if this effective address could be stored for later use. For example, suppose during the course of a program it was necessary to use a memory location specified by indexed addressing more than once, then, rather than use the same indexed addressing mode, it would be better to re-use the effective address calculated the first time. This is where the LEA (Load Effective Address) instruction comes The LEA instruction can only be used with the indexed addressing mode and its action to store the resulting effective address in one of the pointer registers. For example,

```
LEAY 3,X
```

will load the Y register with the effective address calculated by 3,X. In other words, after the instruction the Y register will contain the value in the X register plus 3. You can use the LEA instruction to store an effective address, calculated using a given pointer register, back into itself. For example,

```
LEAX 3,X
```

will store the effective address obtained by adding 3 to the current value of the X register back into the X register. A common use of the LEA instruction is as an increment or decrement of a pointer register. For example, another way to write the program that clears 100 memory locations is,

```
        CLRA
        LDX     @START

@LOOP   CLR     0,X
        LEAX    1,X
        INCA
        CMPA    #99
        BLE     @LOOP
```

where the LEAX instruction adds one to the X register each time through the loop. Notice that the LEA instruction can be used with any of the indexed addressing modes. So for example,

```
LEAX    ,S+
LEAY    A,X
LEAU    D,U
```

are all valid LEA instructions. Notice that LEAX 1,X and LEAX -1,X are the pointer register equivalents of INC and DEC.

The ABX instruction

The ABX (Add B to X) instruction is not really in the mainstream of 6809 instructions in that it singles out the X register for special treatment! The effect of the ABX instruction is to add the contents of the B register considered as a simple binary number to the X register. You might think that ABX is the same as LEAX B,X. However, there is an important difference. The LEAX B,X instruction treats the contents of the B register as a two's complement number and so the range of the effective address is X-128 to X+127 but the ABX instruction treats the contents of the B register as a simple binary number and this give a range of X to X+255.

Program counter relative

'Program counter relative addressing' is an advanced form of indexed addressing that you can easily use without understanding. For this reason, you might like to postpone reading this section until you are entirely happy with the other indexed addressing modes.

In the chapter on branching, it was briefly mentioned that one of the advantages of relative addressing was the fact that a program that used nothing but relative addressing could be moved about in memory and run without having to be re-assembled. The trouble is, that while branch and branch-to-subroutine instructions use relative addressing, instructions that manipulate data actually quote the address of the memory location that they are going to use. If you want to write a fully position-independent program then there has to be a way of using relative addressing with any instruction. The similarity between constant offset indexed addressing and relative addressing has already been noted. The only difference between relative addressing and constant offset indexed addressing is that relative addressing adds the offset to the PC register and indexed addressing adds the offset to

one of the pointer registers. The PC register is not a general purpose pointer register but, to make relative addressing available to all 6809 operations, it can be used in a constant offset indexed addressing mode called 'PC relative'. For example,

```
LDA 5,PC
```

would access the memory location five memory locations further on from the start of the next instruction (see relative addressing in Chapter Seven). In the same way that the BASIC assembler automatically calculates two's complement offsets for relative addressing, most assemblers will calculate the correct offset for P C relative from the value of an address label. So -

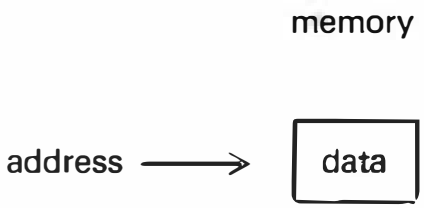
```
LDA @DATA,PC
```

would calculate the offset required to load the A register from the memory location labelled by @DATA,

Using PC relative addressing to produce position independent programs is not a difficult technique but it is better to concentrate on writing assembly language programs that work before venturing into producing more sophisticated programs. For this reason, P C relative addressing will not be built into the BASIC assembler and won't play any part in forthcoming examples. This facility is to be found in both of the commercially available assemblers detailed in Appendix II.

Indirection

Like the last section, this one deals with a topic that can be left until you are ready to tackle something new! 'Indirection' is, in principle, a simple idea that can get a little complicated in practice! The idea of using an address to specify a memory location is something that you should already be thoroughly familiar with. However, in the 6809, an address is nothing more than a 16-bit simple binary number and so it can be stored in two memory locations just like any other 16-bit number. The idea behind indirection is that, instead of giving the address of the memory location that holds the data, you supply the address of a pair of consecutive memory locations that contain the address of the memory location that contains the data! This idea is easier to understand than it is to describe! If simple addressing is imagined as,



then indirect addressing can be depicted as:

Indirect addressing is sometimes expressed as "giving the address of the address". Once you see the general idea you will not be surprised to learn that you can apply indirection more than once. If first level indirection is giving the address of the address then second level indirection is 'giving the address of the address of the address'. And so on to third level indirection and so on to .....! Perhaps fortunately, the 6809 will only handle one level of indirection in an address field. In particular, you can only use indirect addressing in conjunction withhextended addressing and indexed addressing. The usual way of indicating indirection is to enclose the address field in square brackets but as these are not easy symbols to type on the Dragon we will use () brackets. For example,

```
LDA $4000
```

means load the A register with the contents o f memory location $4000 but

```
LDA ($4000)
```

means that $4000 and $4001 contain the address that the A register should be loaded from. When indirection is used with indexed addressing the principle is that the effective address is calculated first (ignoring the indirection) and then this effective address is used as the address of the memory location where the actual address is stored. For example,

```
LDA (4,X)
```

will first add 4 to the contents of the X register to get the effective address and then use this as the address of the pair of memory locations that hold the address of the data.

Indirect addressing is available on all of the indexed addressing modes apart from auto increment by 1 and auto decrement by 1. That is,

```
STA  (,X++)
```

and

```
STA  (,--X)
```

are perfectly good but

```
STA  (,X+)
```

and

```
STA  (,-X)
```

are both invalid. The reason for this restriction is not difficult to see. If a pointer register contains the address of a 16-bit address stored in two memory locations, what is the purpose of adding or subtracting 1 from it to make it point at half of the address pair?

The reason that indirect addressing is treated in this chapter is that ALL indirect addressing including extended indirect addressing is implemented as a variety of indexed addressing. You will find indirection a valuable tool when you come to write large assembly language programs that are intended for use by other programmers. For example, the Dragon's BASIC ROM contains many uses of indirection. One that is worth mentioning is the indirect jump. If you look at the table in Appendix I you will see that the JMP and JSR instructions can be used with indexed addressing and so

```
JMP (address)
```

and

```
JSR (address)
```

are both valid. You may find it difficult to think of a reason for using indirect jumps but suppose that you were writing a large program consisting of a collection of subroutines that kept changing. As the subroutines changed in size their starting addresses would also change and any program that used the subroutines would have to be updated to take into account their new positions. However, if you followed the simple rule of placing all the subroutine start addresses together in the form of a table - a 'jump table' - at the start of your program and insisting that other programmers used the subroutines by indirect jumps through the table then, as long as you kept the table up-to-date, you could move the start addresses of the subroutines as much as you liked without affecting anyone. This is in fact what is done in most large programs but the use of indirect jump tables and such like really comes under the heading of advanced assembly language programming and be will returned to briefly in Chapter Eleven.

Summary of indexed addressing modes

If you include indirection there is a very wide range of indexed addressing modes available to the programmer and it is worth gathering them together in one place -

Mode	example	indirect example
Constant offset	LDA 5,X	LDA (5,X)
Accumulator offset	LDA D,Y	LDA (D,Y)
Auto increment by 1	LDA ,S+	not valid
Auto increment by 2	LDA ,Y++	LDA (,Y++)
Auto decrement by 1	LDA ,-Y	not valid
Auto decrement by 2	LDA ,--U	LDA (,--U)
PC relative	LDA3,PC	LDA (3,PC)
Extended indirect	------	LDA ($400)

These are all of the possible indexed addressing modes. Notice that you cannot combine modes to obtain new addressing modes. For example, LDA 3,X++ is illegal because auto increment can only work without an offset, LDA -X+++ is illegal because you can only auto increment by 1 or 2 and

LDA ,X-- is illegal because you can only auto decrement before the effective address is used.

Machine code details of indexed addressing

You can use indexed addressing without ever worrying about how it is implemented in 6809 machine code but if you are at all interested in understanding how an assembler works, or if you are interested in producing efficient code, then it does help to know how an instruction like LDA 5,X is assembled. In fact the constant offset indexed mode can be assembled into machine code in four different ways depending on the range of the offset.

The fact that an instruction is using indexed addressing is conveyed in its machine code in the same way as extended, immediate or direct addressing. For example, $86 is the code for LDA using immediate addressing and $A6 is the code for LDA using indexed addressing, but which FORM of indexed addressing? The additional information about which of the many variations of indexed addressing is contained in an additional code that follows the machine code. This is called the 'post byte'. In other words, an instruction using indexed addressing uses two memory locations, one to store its instruction code and the following one to supply information about the actual form of indexing being used. The format of the post byte can be seen in the Indexed Addressing Modes Table in Appendix I. The coding of an indexed instruction using this table is quite straightforward. For example, LDB A,Y assembles to $E6, $A5. The $E6 is obtained from the Instruction Code Table in Appendix I in the usual way and the $A5 is obtained from the Indexed Addressing Modes Table, taking RR to be 01 for the Y register.

The section of the table dealing with constant offset indexed addressing deserves a closer look. When writing assembly language the offset can be in the range -32768 to + 32767 but always to include a 16-bit two's complement offset would be very wasteful of time and memory. Most constant offset indexed addresses are of the form 0,X or 4,X, that is the offsets are usually small. For this reason the machine code form of the constant offset indexed address depends on the size of the offset. If the offset is zero or in the range -16 to + 15 it can be included into the post byte giving a very fast, very short instruction. If it is in the range -128 to + 127 the offset is too large to be included in the post byte and so it has to be stored in the next memory location giving the resulting indexed instruction the following form

machine code | post byte | 8 bit offset |

If the offset is in the range -32768 to 32767 then there is no choice but to follow the offset with two memory locations holding a full 16-bit offset. This gives the indexed instruction the following form -

machine code | post byte | offset 1 | offset 2|

(where offset 1 holds the most significant byte and offset 2 holds the least significant byte.) Notice that a constant offset indexed addressing mode instruction can occupy as little as two memory locations or as many as five if the machine code itself occupies two of them. Also notice that there is more than one way to write an instruction involving a small offset. For example, LDA 0,X can be assembled as a zero offset instruction into two bytes, as a five-bit offset instruction in two bytes, as an eight-bit offset in three bytes or even as a 16-bit offset in four bytes! In most cases a good assembler will choose the most economical form of any indexed addressing mode and leave the programmer to worry about more interesting things.

Direct addressing and the DP register

The only addressing register yet to be described is the DP or 'Direct Page register'. When direct addressing was first introduced it was described as a method of addressing memory from 0 to 255. The question is why should the 256 memory locations starting at zero be so favoured by direct addressing? The answer is obviously that a direct address is only eight bits long and this gives a range of 0 to 255. But this is missing the point that an address is a 16-bit number, and so to select one memory location the eight bits specified in direct addressing are extended to 16 bits by adding eight zeros. In other words -

LDA > $31

is interpreted as an instruction to load the A register from $0031. (Recall that > is used to indicate direct addressing - see Chapter Four, "Changing the BASIC assembler into a two-pass assembler".) The $00 that is written in front of the direct address is in fact stored in the DP register. In other words, the DP register holds the most significant eight bits of a direct address. If the DP register was loaded with something other than $00 then a direct address would specify a memory location outside the range 0 to 255. There isn't a

LDDP instruction and so the only way to modify the DP register is by the TFR or EXG instructions. For example, following

```
LDA #$04
TFR A,DP
```

the DP register contains $04 and any direct addresses refer to the range $0400 to $04FF. In other words an instruction like STA >$32 will store the contents of the A register in memory location $0432, the most significant eight bits coming from the DP register and the least significant eight bits coming from the direct address.

This use of the DP register allows direct addressing to be used to access any 256 byte block of memory starting at an address of the form $XX00 where $XX is stored in the DP register. The DP register can significantly reduce the amount of space that a program takes up in memory and can even make it run a little faster but these are not normally considerations that trouble the Dragon assembly language programmer. The best advice is to leave the DP register alone as the BASIC ROM occasionally uses it!

Adding the addressing registers to the BASIC assembler

There are three parts to adding the addressing registers to the BASIC assembler - the 16-bit register instructions such as LDY etc, indexed addressing modes and the EXG and TFR instructions. Adding the rest of the 16-bit register commands is simply a matter of adding the appropriate DATA statements and making allowance for 16-bit immediate data (as for the D register). Adding the indexed addressing modes is a little more tricky in that there are so many different forms to indexing. To keep things as simple as possible only 16-bit constant offset, accumulator offset and auto increment/decrement indexed addressing will be implemented. Thus the BASIC assembler ignores indirection and P C relative addressing. This is not too much of a restriction in that both forms are best left for more advanced programming. Notice also that the only form of constant offset that is allowed is 16-bit offset. This means that instructions like LDA 0,X that could be assembled into two memory locations will in fact takes four but this waste of space is worth the simplification it brings to the BASIC assembler. The instructions EXG and TFR can be adding by included their DATA statements and extending the way that the address field is handled to allow for

instructions like EXG A,B. Which registers are to be exchanged or transferred is indicated by the byte following the machine code for the instruction according to the following table -

```
D = 0    X = 1    Y = 2    U = 3    S = 4
PC = 5   A = 8    B = 9    CC = 10  DP = 11
```

where the code for the source register is stored in the most significant four bits and the code for the destination register is stored in the least significant four bits. So, for example, TFR U,S would assemble to -

```
$1F $34
```

where $1F is the machine code for TFR and $34 is the code for the U register ($3) and the S register ($4) is indicated in the table given above.

```
1 REM BASIC ASSEMBLER V9.1


81 DATA LDX,&H8E,&H9E,&HAE,&HBE,-1
82 DATA LDY,&H108E,&H109E,&H10AE,&H10BE,-1
83 DATA LDS,&H10CE,&H10DE,&H10EE,&H10FE,-1
84 DATA LDU,&HCE,&HDE,&HEE,&HFE,-1
85 DATA STS,-1,&H10DF,&H10EF,&H10FF,-1
86 DATA STU,-1,&HDF,&HEF,&HFF,-1
87 DATA STX,-1,&H9F,&HAF,&HBF,-1
88 DATA STY,-1,&H109F,&H10AF,&H10BF,-1
89 DATA CMPS,&H118C,&H119C,&H11AC,&H11BC,-1
90 DATA CMPU,&H1183,&H1193,&H11A3,&H11B3,-1
91 DATA CMPX,&H8C,&H9C,&HAC,&HBC,-1
92 DATA CMPY,&H108C,&H109C,&H10AC,&H10BC,-1
93 DATA LEAS,-1,-1,&H32,-1,-1
94 DATA LEAU,-1,-1,&H33,-1,-1
95 DATA LEAX,-1,-1,&H30,-1,-1
96 DATA LEAY,-1,-1,&H31,-1,-1
97 DATA ABX,-1,-1,-1,-1,&H3A
98 DATA EXG,&H1E,-1,-1,-1,-1
99 DATA TFR,&H1F,-1,-1,-1,-1
199 DATA ZZZ,-1,-1,-1,-1,-1
```

```
5000 IF M$="EXG" OR M$="TFR" THEN GOSUB 5100 ELSE
GOSUB5500

5100 FOR K=J TO LEN(A$(I))
5110 IF MID$(A$(I),K,1)="," THEN GOTO 5130
5120 NEXT K
5130 L$=MID$(A$(I),K-2,2)
5140 GOSUB 5200
5150 AF$=L$
5160 L$=MID$(A$(I),K+1,2)
5165 IF RIGHT$(L$,1)=" " OR LEN(L$)=1 THEN L$="
" +LEFT$(L$,1)
5170 GOSUB 5200
5180 AF$="&H"+AF$+L$
5185 TYPE=1
5190 RETURN
5200 IF L$=" D" THEN L$="0"
5210 IF L$=" X" THEN L$="1"
5220 IF L$=" Y" THEN L$="2"
5230 IF L$=" U" THEN L$="3"
5240 IF L$=" S" THEN L$="4"
5250 IF L$="PC" THEN L$="5"
5260 IF L$=" A" THEN L$="8"
5270 IF L$=" B" THEN L$="9"
5280 IF L$="CC" THEN L$="A"
5290 IF L$="DP" THEN L$="B"

5800 TYPE=3
5810 L$=MID$(A$(I),K-1,1)
5820 OF=0
5830 IF L$="A" THEN OF=&H86
5840 IF L$="B" THEN OF=&H85
5850 IF L$="D" THEN OF=&H8B
5860 L$=MID$(A$(I),K+1,1)
5870 IF L$="-" THEN L$=MID$(A$(I),K+2,1):OF=&H82
5880 IF L$="-" THEN L$=MID$(A$(I),K+3,1):OF=&H83
5890 RF=0
5900 IF L$="Y" THEN RF=1
5901 IF L$="U" THEN RF=2
5902 IF L$="S" THEN RF=3
```

```
5910 IF MID$(A$(1),K+2,1)="+"THEN OF=&H80
5920 IF MID$(A$(I),K+2,2)="++" THEN OF=&H81
5930 IF OF<>0 THEN AF$=STR$(RF+OF):RETURN
5950 OF=&H89+RF
5955 AF$=" "+AF$
5960 A=VAL(AF$)
5970 IA=1
5980 IF A>=0 THEN RETURN
5990 AF$=STR$(65536+A)
5995 RETURN

6035 IF IA=1 THEN PRINT#-PRT,HEX$(OF);

6241 IF TYPE=1 AND RIGHT$(M$,1)="X" THEN TYPE=4
6242 IF TYPE=1 AND RIGHT$(M$,1)="Y" THEN TYPE=4
6243 IF TYPE=1 AND RIGHT$(M$,1)="U" THEN TYPE=4
6244 IF TYPE=1 AND RIGHT$(M$,1)="S" THEN TYPE=4

6255 IF IA=1 THEN POKE P,OF:P=P+1:TYPE=4
```

Subroutine 5800 processes indexed addressing by making up the code for the post byte in accordance with the Indexed Addressing Modes Table in Appendix I Subroutine 5100 handles the coding of the byte following the EXG or TFR instruction indicating which pair of registers are involved. Notice that subroutine 6000 has to be modified to print and POKE the correct number of memory locations for a 16-bit constant offset.

The processing of the address fields for both indexed addressing and the EXG and TFR instructions is very crude and doesn't allow for blanks included in the instruction. So, for example, you must write EXG A,B rather than EXG A, B and LDA 0,X rather than LDA 0 ,X.

A general multiple-precision arithmetic subroutine

Before moving on to practical examples involving the Dragon's sound there is some unfinished business concerning multiple-precision arithmetic, dealt with in Chapter Eight, that needs clearing up. To add two numbers together, each @N bytes long, the first stored with its most significant byte at

@NUM1 and the second stored with its most significant byte at @NUM2 and store the answer starting at @ANS use -

```
@ADD     LDB     @N
         LDX     #@NUM1
         LEAX    B,X
         LDY     #@NUM2
         LEAY    B,Y
         LDU     #@ANS
         LEAU    B,U
         ANDCC   #$FE

@A LOOP  LDA     0,-X
         ADCA    0,-Y
         STA     0,-U
         DECB
         BNE     @ALOOP
         RTS
```

To subtract the two numbers simply change the ADCA instruction to SBCA. Notice the way that the three pointer registers X,Y and U are used with auto decrement to 'step through' the memory locations of each number. Also notice the way that the C bit is cleared before the first add so that the ADC instruction can be used to add the first memory locations without error. The LEA instructions at the start of the program adjust the pointer registers to point to one memory location before the least significant byte of the numbers because the auto decrement happens before the effective address is used.

Using Dragon Sound

Although you might not think so from the limited beeping that BASIC restricts you to, one of the most flexible features of the Dragon is its sound. This section is concerned with generating sounds on the Dragon and provides plenty of opportunity to use indexed addressing.

The Dragon's sound hardware is fully described in the companion volume to this book, "The Anatomy of the Dragon". However, the only essential detail from the point of view of the following assembly language subroutines is that, after a few operations concerned with initialisation, storing a number

in memory location $FF20 will produce a voltage proportional to the number which is fed to the speaker of the TV set via a UHF modulator. To be more exact, only b7 to b2 of the location actually affects the output. What this means is that you can produce a voltage that varies from 0 to the maximum level in 64 equal steps by storing a number in the range 0 to 63 in bits b7 to b2 of memory location $FF20, Using just this simple fact it is possible to create an almost unlimited range of sounds.

There are three ways in which a steady tone can vary, in volume, in pitch and in quality. All steady tones are the result of periodic waveforms. The volume is related to the amplitude of the waves, the pitch to the rate of repetition and the quality to the shape of the wave. For example, the purest of all tones is a perfect sine wave (see fig 9. 1). A particularly rough sounding tone is produced by the 'sawtooth' wave form (see fig 9.2). To produce the sawtooth wave form all that is necessary is to store a series of numbers that increase to some maximum and then reduce to zero and so on in location $FF20. Obviously to create tones with a given quality it would be useful to have a program that allowed the user to specify a series of numbers and then hear the tone that they produce. An assembly language program to do this is quite easy to write, the only difficult part being to find a good way of letting the user specify the series of numbers. This is a task that is better suited to BASIC and so in this section the use of assembler together with BASIC will be examined.

The following assembly language program will take the series of values stored in memory starting at $7F00 and then repeatedly store them in memory location $FF20 so that you can hear the quality of the sound they produce. This part of the program is fairly easy but to provide flexibility, the program has to allow the user to decide how many values are specified and how long the delay should be between storing each value. The number of values is stored at $7EFF and the delay at $7EFE. Also, if the program is going to be used as part of a BASIC program for designing sounds, there should be some way to determine how long the sound should last. The sound duration is thus specified in memory location $7EFD in fiftieths of a second so that it can be compared with the Dragon's TIMER clock.

```
@SOUND    EQU    $FF20
@TABLE    EQU    $7F00
@NUM      EQU    $7EFF
@PITCH    EQU    $7EFE
@DUR      EQU    $7EFD
```

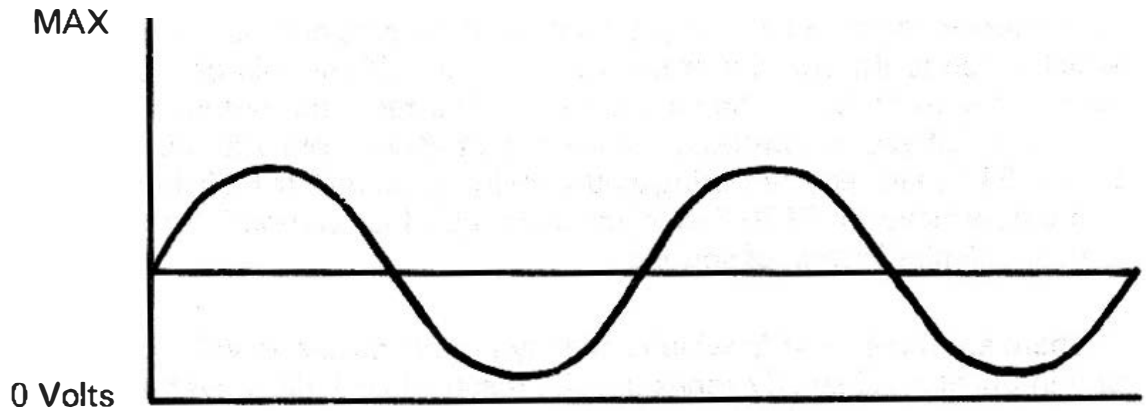


Fig 9.1 A sine wave: the purest of tones

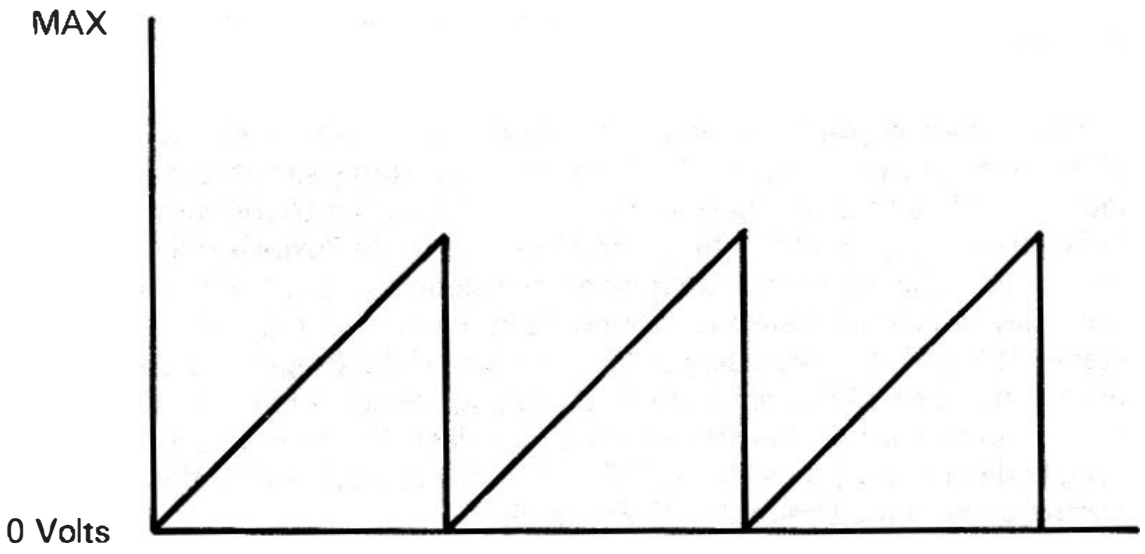


Fig 9.2 A saw tooth wave form — a rough sound

```
@TIMER    EQU     $113

@START    BSR     @INIT
          CLR     @TIMER
@REP              LDX @TABLE
          CLRA
@LOOP     LDB     A,X
          ASLB
          ASLB
          STB     @SOUND
          BSR     @DELAY
          INCA
          CMPA    @NUM
          BLO     @LOOP
          LDA     @DUR
          CMPA    @TIMER
          BHI     @REP
          RTS

@INIT     LDA     $FF23
          ORA     #8
          STA     $FF23
          LDA     $FF01
          ANDA    #$F7
          STA     $FF01
          LDA     $FF03
          ANDA    #$F7
          STA     $FF03
          RTS

@DELAY    LDB     @PITCH
@LDEL     DECB
          BNE     @LDEL
          RTS
```

Subroutine @INIT sets up the Dragon's sound channel so that you can hear the results of the program and @DELAY is a typical delay subroutine as described in Chapter Eight. The rest of the program is concerned with moving the values in memory to $FF20 over and over again for the correct duration.

Notice the way that the A register is used to determine both which memory location will be transferred to $FF20 and when all of the locations have been transferred.

If you just EXEC this program you the results that you get will depend on whatever happens to be stored in the memory locations used for its data. To make the program useful it has to be used with a BASIC program that sets the data area to something appropriate. A machine code program can be saved on tape using

```
CSAVE "filename",start,end,transfer
```

THis command will save on tape the contents of memory from 'start' to 'end' (where 'transfer' = end - start). To save the sound program use -

```
CSAVE "SOUND",&H7000,&H7030,&H31
```

After saving the sound program delete the BASIC assembler using NEW and type in the following program -

```
10 CLEAR 1000,&6FFF
20 INPUT "HOW MANY VALUES";V
30 IF V>255 THEN GOTO 20
40 POKE &H7EFF,V
50 FOR I=1 TO V
60 PRINT "VALUE ";I;"=";
70 INPUT A
80 IF A>255 THEN GOTO 60
90 POKE &H7F00+I-1,A
100 NEXT I
110 INPUT "PITCH";P
120 IF P>255 THEN GOTO 110
130 POKE &H7EFE,P
140 INPUT "DURATION";D
150 IF D>255 THEN GOTO 140
160 POKE &H7EFD,D
170 EXEC &H7000
180 GOTO 110
```

then load the sound program using -

```
CLOADM "SOUND",0
```

When the BASIC program is RUN it will use the machine code to let you hear the quality of the note produced by the list of values that you type in.

Altering the quality of the sound using the waveform works well for steady tones but most sounds contain a range of pitches. For example, a typical 'laser zap' in a space game will start off at a high pitch and descend to a low pitch. This is fairly easy to do if the wave form is kept simple and the simplest waveform to program is a square wave.

```
@LASER    BSR     @INIT
          LDB     #1
@LOOP     LDA     #$FC
          STA     $FF20
          BSR     @DELAY
          CLR     $FF20
          BSR     @DELAY
          INCB
          CMPB    #180
          BNE     @LOOP
          RTS


@DELAY    TFR     B,A
@DLOOP    DECA
          BNE     @DLOOP
          RTS
```

The B register is used to determine the length of the delay produced by @DELAY which increases by one each time through the program.

Steady tones and tones that change in pitch are not all that the Dragon can produce. Many special effects are based upon 'white noise'. White noise, a sound rather like that made by a radio between stations corresponds to a jumbled non-repeating wave form (see fig 9.3). To produce this sort of wave form requires a source of numbers that are as good as random. The trouble is that it takes rather too long to generate random numbers using the BASIC RND function. An alternative source of varied numbers that provides a fair

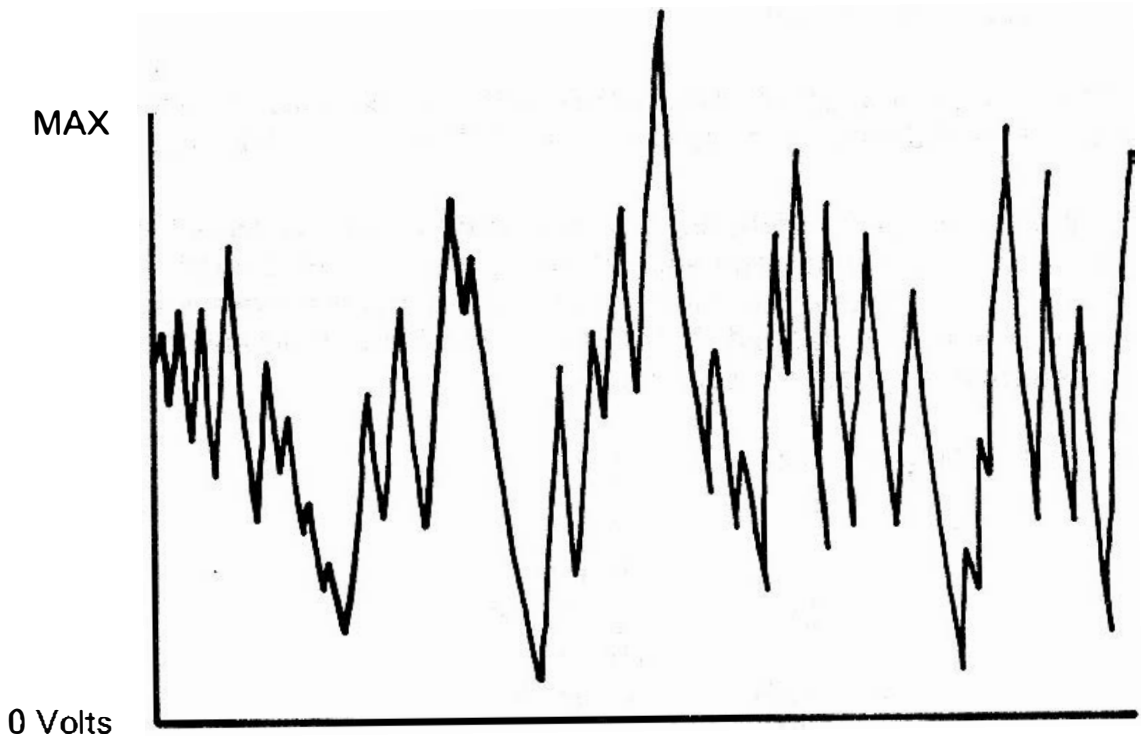


Fig 9.3 White noise

approximation to a random sequence is the BASIC ROM! The following program allows you to listen to the sound of BASIC -

```
@START    EQU     $9000
@END      EQU     $B000

@WHITE    BSR     @INIT
          LDX     #@START
@LOOP     LDA     ,X+
          BSR     @SOUND
          CMPX    #@END
          BNE     @LOOP
          RTS

@SOUND    STA     $FF20
@DELAY    LDA     #20
@DLOOP    DECA
          BNE     @DLOOP
          RTS
```

You can send different sections of the ROM to the sound generator simply by changing @START and @END.

To give you some idea of how to use white noise the following program generates a 'gunshot' effect by progressively fading down the white noise -

```
@START    EQU     $9000

@CRAK     BSR     @INIT
          CLR     @VOL
          LOX     #@START
@LOOP     LOA     ,X+
          ANDA    @VOL
          BSR     @SOUND
          LOA     ,X+
          ANDA    @VOL
          BSR     @SOUND
          LDA ,X+
          ANDA    @VOL
          BSR     @SOUND
          DEC     @VOL
          BNE     @LOOP
          RTS
@VOL      FCB 0
```

Notice the way that the volume is faded down by ANDing the number with the contents of @VOL.

Summary

1) There are five addressing registers -

the two 16-bit index registers X and Y
the two 16-bit stack pointers U and S
and the eight bit direct page register DP

2) Both the index registers and the stack pointers can be used for indexed addressing.

3) There are three indexed addressing modes -

constant offset indexed
accumulator offset indexed
auto increment/auto decrement indexed

4) Indirect addressing specifies the address of the address that is to be used in an operation.

5) Indirection can be used with any indexed addressing mode apart from auto increment and auto decrement by one.

6) Indirect extended addressing is also available and is implemented as a special indexed addressing mode.

7) The DP register holds the most significant byte of a direct address.

8) The TFR and EXG instructions can be used to move data between all of the 6809's registers.

Micro projects

1) In Chapter Eight a delay subroutine was given that used the D register. Re-write it using one of the pointer registers as a counter.

2) Write a short BASIC program that will use the SOUND subroutine given earlier to produce white noise by POKEing random numbers in the sound table (starting at @TABLE). Use RND to generate the numbers (in the range 0 to 63) and try experimenting with the values of pitch and duration.

Chapter Ten

The Stack Pointers and Interrupts

A 'stack' is one of the most useful ways of storing temporary data. The 6809 uses a stack to hold the return address following a JSR or BSR instruction. As well as this implicit use of a stack by JSR and BSR instructions, assembly language programs can make more direct use of stacks to store data. The subject of stacks and how they are used brings us to a consideration of 'interrupts'. Interrupts are the main way that a computer can be made to respond to the outside world. In the case of the Dragon, the most interesting use of interrupts is in providing the BASIC TIMER facility. In this chapter both general stack operations and interrupts are described. The chapter closes with an example of how an interrupt routine can improve the Dragon's keyboard.

A stack

A stack is simply an area of memory used in conjunction with a stack pointer. The stack pointer contains the address of the item of data stored in the stack. The normal method of storing data on the stack is via the PSH (PuSH) operation which subtracts one from the stack pointer and then stores the data in the memory location that it points at. The normal method of removing data from the stack is via the PUL (PULl) instruction which accesses the item that the stack pointer is pointing at and then adds 1 to the stack pointer. Notice that for a PSH operation the stack pointer moves down one BEFORE the item is stored but for a PUL operation the stack pointer moves up one AFTER the item has been accessed.

The PSH and PUL operations on the stack produce a 'Last In First Out' or 'LIFO' effect. For example, if you push three numbers on to the stack in the order 1, 2, 3 the stack will look like this (SP is the stack pointer):

Carrying out a PUL on the stack will retrieve 3 and leave the stack pointer pointing as 2. Thus a second PUL will retrieve the value 2 and a third, the value 1. The numbers went in 1, 2, 3 but came out 3, 2, 1. That is, the last number onto the stack came out first.

The 6809 stack pointers U and S

The fact that the S and U addressing registers are also stack pointers was mentioned in Chapter Nine. The S register is so called because it acts as the stack pointer for the 'System stack'. The system stack can be used to store temporary data generated by a program but it is also used automatically by the 6809 to store temporary data generated in the course of running your program. In this sense the system stack has to be shared with the 6809. The U register is so called because it acts as the stack pointer for the 'User stack'. The user stack isn't used by the 6809 and is free for any assembly language program to use as required. The only trouble is that subroutines inside the BASIC ROM may use the U stack, so applications program have to take care if they are going to use any part of the BASIC ROM.

The basic stack operations on the S and U registers are PSHS, PULS, PSHU and PULU. Each operation can PSH or PUL any of the 6809's registers. For example, the PSHS instruction take the form -

```
PSHS 'register list'
```

where 'register list' is a list of the names of the registers to be pushed onto the S stack. So -

```
PSHS X,Y,A
```

will push the contents of the X, Y and A registers onto the S stack. The other three instructions, PULS, PSHU and PULU, can also pull or push a list of registers. For example,

```
PULS X,Y,A
```

will restore the values that were pushed onto the S stack by the PSHS X,Y,A instruction to the X,Y and A registers - as long as nothing else has been pushed onto the stack in the mean time. The only restriction on the register list is that you cannot pull or push the S register onto the S stack nor the U register onto the U stack.

You might be wondering about the order that the registers are pushed or pulled. The order that you write the registers in the register list doesn't affect the order that the registers are pushed or pulled. For example,

```
PSHS X,Y
```

is the same as

```
PSHS Y,X
```

In fact the order that the registers are pushed on to the stack is strictly predetermined by the following priority -

1	PC
2	U or S
3	Y
4	X
5	DP
6	B
7	A
8	CC

From a list of registers that are to be pushed onto the stack the registers corresponding to the lowest numbers are pushed first. For example, in

```
PSHS CC,A,X
```

the X register is pushed first, then the A register then the CC register. In other words PSHS CC,A,X is the same as

```
PSHS X
PSHS A
PSHS CC
```

The order in which registers are pulled from the stack is the reverse of the order in which they are pulled. That is, the registers in a register list will be pulled so that the highest numbered registers are pulled first. So

```
PULS X,B,CC
```

is the same as

```
PULS CC
PULS B
PULS X
```

Apart from the PSH and PUL instructions, there are no other stack operations. However, as the stack pointers are both general pointer registers, values on the stack can be manipulated using indexed addressing. For example,

```
PSHSA
ADDA,S+
```

doubles the value in the A register by first pushing its value on the stack and then adding it back into the A register. Notice the way that auto increment on the S register returns it to its original value before the PSHS - thus 'cleaning up' the stack. It is important that, if you use a stack to store temporary data, you remove it and leave the stack as you found it. Otherwise you could find some odd things happening. In particular, if you push more onto the stack than you pull off the stack will eventually grow to occupy all of the memory! Notice that it is in general necessary to allocate sufficient memory to a stack so that it doesn't overflow into areas of memory that are being used for other purposes. In the Dragon the S stack is usually initialised by the system to be just below the temporary string storage area used by BASIC.

Subroutines and the system stack

The two stack pointers U and S are identical in their use apart from the way that the 6809 automatically uses the system stack to store, among other things, the return address following a subroutine call. Whenever you use BSR or JSR the address of the next instruction is automatically stored on the system stack. The action of an RTS instruction is to pull two bytes off the stack and place them in the PC register which, if everything has gone to plan, should return control to the instruction following the branch. In other words RTS has the same effect as PULS PC. This use of a stack to store the return address has the advantage that a subroutine can call another subroutine and, because of the Last In First Out property of a stack, the return addresses come back in the correct order. The only thing that can go wrong with this mechanism is that the subroutine stores more on the stack than it takes off and then the RTS instruction will simply pull rubbish off the stack with predictable disastrous consequences.

Interrupts

The idea of an interrupt is so familiar to humans that it is hardly worth a second thought. If you are reading a book and the telephone rings you would have no difficulty in marking your place in the book, answering the phone and then, after the call is complete, returning to the marked place in the book as if nothing had happened. Contained in this description are the essential elements of all interrupt handling. First there is a signal from the outside world - the interrupt. As a result of this interrupt the current preoccupation is suspended but enough information is stored to enable the task to be restarted after the interrupt has been dealt with. The interrupt is dealt with and then the original occupation is restored.

The 6809 can respond to three different types of interrupt - the NMI (Non-Maskable Interrupt), IRQ (Interrupt ReQuest) and FIRQ (Fast Interrupt ReQuest). Each of these types of interrupt corresponds to a physical connection to the 6809 chip inside the Dragon. A signal on one of these connections indicates a request to interrupt the 6809 from whatever it is currently doing. Exactly what happens following an interrupt depends on which source caused the interrupt.

IRQ Following a signal on the IRQ line the 6809 completes the instruction that it is carrying out, then it stacks all of the registers and jumps to the location whose address is stored in $FFF8 and $FFF9. In other words, after

stacking all the registers, the6809 executes a JMP ($FFF8). In the Dragon the address $FFF8 is shifted by hardware down into the BASIC ROM at $BFF8 which contains the address $010C. This means that an IRQ interrupt transfers control to $010C. Details of what the Dragon uses the IRQ interrupt for will be given later.

NMI Following a signal on the NMI line the 6809 completes the instruction that it is carrying out then it stacks all of the registers and jumps to the location whose address is stored in $FFFC and $FFFD. In the Dragon the address $FFFC is shifted by hardware down into the BASIC ROM at $BFFC which contains the address $0109.

FIRQ Following a signal on the FIRQ line the 6809 completes the instruction that it is carrying out and then stacks the PC register and the CC register and then jumps to the location whose address is stored in $FFF6 and $FFF7. Notice that in this case not all of the registers are stacked following a FIRQ interrupt. It is this that makes it a fast interrupt. In the Dragon the address $FFF6 is shifted by hardware down into the BASIC ROM at $BFF6 which contains $010C. The FIRQ is only used in the Dragon to detect the presence of a ROM program cartridge.

3 In general, the6809's interrupts can be used in a wide variety of ways but the Dragon's hardware has been designed to put interrupts to good use and so they are dedicated to a single purpose. In practice, the only interrupt that is of any interest to the Dragon assembly language programmer is the IRQ interrupt that is used to provide the BASIC TIMER function. A practical example of how this interrupt can be used is given at the end of this chapter but for now some of the 6809 instructions concerned with interrupts in general will be described.

The RTI instruction

The three types of interrupt have one thing in common they all save some of or all of the registers on the system stack and then do an indirect jump through a fixed location. The destination of the jump is a program usually referred to as an 'interrupt handler'. What exactly the interrupt handler does depends very much on what caused the interrupt. For example, in the case of the Dragon the IRQ handler adds one to the current value of memory locations used for the timer. Once the interrupt handler has finished control

has to be returned to the program that was interrupted. This is done by using the RTI (ReTurn from Interrupt) instruction which is to an interrupt what the RTS instruction is to a subroutine. The RTI not only returns control to the program that was interrupted, it also restores to their original registers any values that were pushed onto the stack by the interrupt. If all of the registers were pushed onto the stack then following an RTI the 6809 is back to its original condition before the interrupt even if the interrupt handler used some of the registers. However, if the interrupt only saved some of the registers, then the interrupt handler has to be careful not to alter any of the registers that are not going to be restored by the RTI instruction.

Condition codes and interrupts

The condition code register was introduced in Chapter Seven in connection with the branch instructions. However, there are a number of condition code bits that are concerned with interrupts and the way that the system stack is used. The full format of the CC register is -

b7	b6	b5	b4	b3	b2	b1	b0
E	F	H	I	N	Z	V	C

The H, N, Z and C bits have already been described in Chapter Seven. The E or Entire bit is used by the RTI instruction to discover how many registers were pushed onto the stack by the interrupt and so how many registers should be pulled off the stack before returning control to the program that was interrupted. If the E bit is 1 then all (i.e. the entire set) of the registers were pushed and so the RTI results in

```
PULS CC,A,B,DP,X,Y,U,PC
```

If the E bit is 0 then only the CC and PC register are pulled from the stack -

```
PULS CC,PC
```

There is a subtle point here. As the RTI instruction pulls the CC register off the stack, which value of the E bit does it take notice of? The answer is that it first restores the CC register by pulling it off the stack and then it examines the E bit to discover how much more has to be pulled off the stack. This means that an interrupt handler can change the CC bits as much as it likes and the RTI

instruction will get it right. You can in fact manipulate the E bit stored on the system stack to change the action of an RTI instruction for special purposes but take great care that you pull the right things of the stack. For example, suppose an IRQ handler didn't use any of the registers that the main program was using. then you could save the unnecessary 'unstacking' of the entire register set by -

```
PULS A
ANDA #$7F
LDS7,S
PSHSA
RTI
```

The first instruction pulls the value that would be returned to the CC register following an RTI and places it in A. The E bit is then set to 0 (by ANDing with $7F), the stack pointer is adjusted to get rid of the unnecessary register values that were stored and then the new value for the CC is PSHed into the correct position on the stack. The following RTI will now only restore the CC and PC registers because the E bit is 0. This sort of trick is only worth using when your are desperate to make a program run faster and it can lead to programs that are very difficult to debug.

The I and F bits are both concerned with stopping interrupts having any effect. The 6809 will only take any notice of IRQ signals if the I bit is 0. Thus the I bit can be used to mask the effects of interrupts, Setting and clearing it can be used to control when an interrupt is allowed. The instruction

```
ANDCC #$EF
```

clears the I bit and

```
ORCC #$10
```

sets it to 1. The F bit will similarly disable FIRQ interrupts. If the F bit is 0 then the 6809 will take notice of signals on the FIRQ line but it will ignore them is F is 1. The instruction -

```
ANDCC #$BF
```

will clear the F bit and

ORCC #$40

sets it to 1.

Apart from enabling the programmer to decide when interrupts should be allowed to happen, the I and F bits are also used by the interrupts themselves to stop the difficulty of an interrupt occurring during the operation of an interrupt handler! The exact action of the IRQ interrupt on the CC register is:

1) The E bit is set to 1 and the CC register is pushed on the system stack

2) The I bit is set to 1 to mask out any further IRQ interrupts

This means that following an IRQ interrupt the interrupt handler will not be interrupted by IRQ again but it will be interrupted by an FIRQ or a NMI. In this sense both FIRQ and NMI have a higher priority than IRQ.

The exact action of a FIRQ interrupt on the CC register is:

1) The E bits is set to 0 and the CC register is pushed onto the system stack

2) The F and bits are boh set to 1 to mask any further FIRQ or IRQ interrupts

Thus a FIRQ interrupt handler will not be interrupted by another FIRQ or a subsequent IRQ unless the CC register is altered. This once again establishes FIRQ as having a higher priority than IRQ. The NMI interrupt behaves like the IRQ interrupt except that it sets both the F and I bits so masking any other interrupts apart from another NMI. A NMI cannot be masked hence its name and in this sense it is a higher priority interrupt than FIRQ and IRQ.

In general there are two ways of stopping an IRQ or FIRQ interrupt. You can stop it at source by altering the device that produces it, or you can set the CC register so as to mask the interrupt. The device that causes the interrupt usually has to be reset in some way or another before it can cause another interrupt. Resetting, and any other operations that the external device requires, are all the responsibility of the interrupt handler. Indeed it is possible that more than one external device can cause an interrupt and in this case the

first job that the interrupt handler has to tackle is to find out what caused the interrupt. As you can imagine, in general use interrupts can become very complicated.

The instructions CWAI and SYNC

The normal use of an interrupt signal is to stop the 6809 from what it is doing and transfer its processing powers to a different and perhaps more urgent task. However, there are occasions when the 6809 has nothing better to do than wait for an interrupt to occur. The two instructions CWAI (Clear and WAit for Interrupt) and SYNC (SYNChronise) are both concerned with making the 6809 suspend its operation until something causes an interrupt. The CWAI #$XX instruction ANDs the CC register with the immediate byte, stacks all of the registers on the system stack and then waits for an enabled interrupt. The immediate byte can be used to select which interrupt signal will be enabled. Notice that the CWAI instruction will cause even the FIRQ interrupt handler to be entered with all the registers stored on the stack. The CWAI instruction can save time in handling interrupts because when the interrupt occurs the registers are already stacked.

The second 'wait for an interrupt' type of instruction is SYNC. A SYNC instruction causes the 6809 to halt processing and wait for an interrupt to occur. If a non-masked interrupt occurs then the usual sequence of register stacking appropriate to the interrupt is completed and the interrupt handler is entered. In this case the SYNC instruction causes the 6809 to wait for an interrupt which is then processed as normal. However, if a masked interrupt occurs then instead of being ignored it removes the 6809 from its 'sync' state and allows it to carry on processing instructions. In this way masked interrupts can be used to synchronise the 6809 to external events. An example of the use of the SYNC instruction is given as a micro project at the end of the chapter.

Software interrupts - SWI, SWI2, SWI3

It may seem strange but the idea of an interrupt is so useful that the 6809 has three instructions that will force the 6809 to behave as if it had received an external interrupt signal! SWI (SoftWare Interrupt), stacks all the registers, sets the I and F bits to mask external interrupts and then cause an indirect

jump to $FFFA. In the Dragon this address is moved using hardware down to $BFFA. The SWI2 and SWI3 also stack all of the registers but they don't mask external interrupts. SWI2 causes an indirect jump through $FFF4 which the Dragon's hardware has moved down to $BFF4 and the SWI3 instruction causes an indirect jump through $FFF2 which the Dragon's hardware has moved down to $BFF2.

The SWI instructions are generally used to implement advanced or special features such as calls to operating systems, machine code debuggers (see Chapter Eleven) and so on.

Adding stack operations to the BASIC assembler

The instructions RTI, CWAIT, SYNC, SWI, SWI2 and SWI3 can all be added to the BASIC assembler simply by including the appropriate DATA statements. However, to accomodate PSH and PUL the handling of the address field has to be extended once again. Both PSH and PUL use the byte following their machine code to store the list of registers to be pushed or pulled according to the following table:

PC	S/U	Y	X	DP	B	A	CC
128	64	32	16	8	4	2	1

The value stored in the following byte is the sum of the codes corresponding to each register to be pushed or pulled. So, for example, the machine code for-

```
PSHS PC,X,A
```

is

```
$34 $92
```

where $34 is the machine code for PSHS and $92 = 128 + 16 + 2 is derived from the above table.

```
1 REM BASIC ASSEMBLER V10.1


100 DATA PULS,&H35,-1,-1,-1,-1
101 DATA PULU,&H37,-1,-1,-1,-1
102 DATA PSHS,&H34,-1,-1,-1,-1
103 DATA PSHU,&H36,-1,-1,-1,-1
104 DATA RTI,-1,-1,-1,-1,&H3B
105 DATA CWAIT,&H3C,-1,-1,-1,-1
106 DATA SYNC,-1,-1,-1,-1,&H13
107 DATA SWI,-1,-1,-1,-1,&H3F
108 DATA SWI2,-1,-1,-1,-1,&H103F
109 DATA SWI3,-1,-1,-1,-1,&H113F


5000 IF M$="EXG" OR M$="TRF" THEN GOSUB 5100
5005 IF LEFT$(M$,3)="PUL" OR LEFT$(M$,3)="PSH" THEN
GOSUB5300 ELSE5500
5300 A=0
5310 FOR K=J TO LEN(A$(I))
5320 L$=MID$(A$(I),K,2)
5330 IF L$="PC" THEN A=A+128:K=K+1
5340 IF L$="DP" THEN A=A+8:K=K+1
5350 IF L$="CC" THEN A=A+1:K=K+1
5360 L$=MID$(A$(I),K,1)
5370 IF L$="A" THEN A=A+2
5380 IF L$="B" THEN A=A+4
5390 IF L$="X" THEN A=A+16
5400 IF L$="Y" THEN A=A+32
5410 IF L$="S" OR L$="U" THEN A=A+64
5420 NEXT K
5430 AF$=STR$(A)
5440 TYPE=1
5450 RETURN
```

The new subroutine 5300 processes the address field following a PSH or a PUL and makes up the value to be stored in the memory byte following the machine code for the instruction according to the previous table.

The Dragon's use of interrupts

We already know that the Dragon's hardware moves all of the memory locations used by the interrupts down into the BASIC ROM but what hasn't yet been mentioned is that you can change the destination of an interrupt. Each of the 6809's interrupts causes an indirect jump through memory locations in ROM down to a series of memory locations in RAM. These RAM memory locations, for those interrupts that are used, contain JMP instructions that finally transfer control to the interrupt handlers. The important point is that as the JMP instructions are stored in RAM they can be changed and so, interrupts can be intercepted on their way to their interrupt handlers. This situation is best summarised by the following table:

Interrupt	indirects through	indirects to	contents
SWI3	$BFF2	$100	0
SWI2	$BFF4	$103	0
FIRQ	$BFF6	$10F	JMP $B469
IRQ	$BFF8	$10C	JMP $9D3D
SWI	$BFFA	$106	JMP $D521
NMI	$BFFC	$109	0
Reset	$BFFE	$B3B4	-----

The meaning of this table should be clear but, to take an example, an IRQ interrupt indirects though $BFF8 which contains the address $010C. Thus, following an IRQ, control passes to $010C which contains the instructions JMP $9D3D which finally transfers control to the interrupt handler. Notice that the table contains an entry for 'Reset' which indicates where the 6809 transfers control to when the reset button is pressed. Some of the 6809's interrupts are not used on the Dragon and this is indicated by the memory locations in the table containing zeros. These unused interrupts could be put to use by applications programs but there is always the possibility that future Dragon systems programs will use one of them with resulting conflict.

The only two interrupts that the standard Dragon uses are FIRQ and IRQ. The FIRQ interrupt is used by a cartridge ROM to gain control from BASIC and start its own program running. The IRQ interrupt is used to produce the 1/50th of a second clock. The way that this works is that the TV frame sync pulse is connected to one of the PIAs which causes an IRQ interrupt every 1/50th of a second. (For more information see "The Anatomy of the Dragon".)

By intercepting the IRQ interrupt, an assembly language program can carry out a fixed action every 1/50th of a second orso. For example, an auto repeat facility can be added to the Dragon's keyboard by setting memory locations $150 to $159 to $FF. While a key is held down each setting of the memory locations cause a single repeat. If the IRQ interrupt is intercepted by an assembly language program that sets the memory locations to $FF and then passes control to the original interrupt handler, then everything will function normally but the keyboard will auto-repeat every 1/50th of a second.

```
@IRQ     EQU     $10D
@AUTO    ORCC    #$10
         LDX     @IRQ
         STX     @TIM
         LDX     #@REP
         STX     @IRQ
         ANDCC    #$EF
         RTS

@REP     LDA     #$FF
         LDX     #$0150
@LOOP    STA     0,X+
         CMPX    #$15A
         BNE     @LOOP
         LDX     @TIM
         JMP     0,X
@TIM     FDB     0
```

The first part of the program @AUTO first disables the IRQ interrupts and then changes the JMP $9D3D to JMP @REP. The @REP subroutine simply sets the memory locations to $FF and then jumps to the original interrupt handler. The address of the original interrupt handler is stored in @TIM and the JMP 0,X instruction transfers control to the the address stored in X. To use this program simple EXEC it once after assembling it and, every 1/50th of a second, it sets the memory locations to $FF and produces an auto repeat. You may find that 1/50th of a second is a little fast for an auto repeat and so the final version of the program produces a repeat every 1/2 of a second by counting the number of interrupts between each setting of the memory locations. The following section needs to be added to the @AUTO part of the previous program:

```
@REP    LDX     @TIM
        DEC     @COUNT
        BEQ     @SET
        JMP     0,X
@SET    LDA     #$FF
        LDX     #$0150
@LOOP   STA     0,X+
        CMPX    #$15A
        BNE     @LOOP
        LDA     #25
        STA     @COUNT
        LDX     @TIM
        JMP     0,X
@TIM    FDB     0
@COUNT  FCB     25
```

Even with this improvement the auto-repeat program needs some work to be useful. Ideally a key should not auto repeat until it has been held down for some minimum time and then it should repeat at quite a fast rate.

Summary

1) The 6809 has two stacks - the system stack using the S register as the stack pointer and the User stack using the U register as the stack pointer.

2) The system stack is use automatically to save the return address following a BSR or JSR instruction and during interrupts to save the values in the registers.

3) Of the three 6809 interrupts only the IRQ interrupt that is used to produce the system clock is likely to be useful to the assembly language programmer working with an unmodified Dragon.

Micro project

1) Use the SYNC instruction to ensure that the squash program given in Chapter Eight only changes the display once every 1/50th of a second. Hint - you only need to modify the @DELAY subroutine.

Chapter Eleven

Assembly Language Style and Practice

Once you have mastered the details of 6809 assembler the only way to extend your skill is to write programs. As with nearly all aspects of computing, practice is essential. Assembler is a powerful language and if a program using a good method doesn't work fast enough in assembler then there is nothing you can do but get a better computer! Although assembler is powerful you still have to know how to solve the problem that you are interested in and in some ways this can be more difficult in assembler than in a high level language. If you have no idea of how to go about solving a problem using BASIC then you have little chance of getting any further with assembler. Despite the fact that there is no way of becoming a proficient assembly language programmer without practice, there are a few guidelines and suggestions that are worth knowing about.

Subroutines

It is well known that the best way to write a large program is to break it down into a collection of smaller programs. It is almost impossible to think of a large program without dividing it down into sections that perform specific tasks and it makes sense to write such a program in a way that reflects these divisions. In BASIC and in assembler the subroutine is the standard way of writing the small 'modules' that fit together to produce a finished program. The use of subroutines in assembler is all the more important because of the very limited operations that the language offers you. In principle whenever you write an assembly language program your objective should be to build up a collection of subroutines that carry out more complex operations. This collection of subroutines can not only be use to implement the current program they form a programmer's 'inheritance' to be used in future

programs. For example, if you are writing a program that needs to carry out any amount of arithmetic then you need to write some subroutines to carry out arithmetic. Perhaps less obviously, if you are writing a games program or a graphics program you should first write subroutines to plot a single point in a given colour and then build up subroutines that plot the shapes that you are using. In this example, notice that the shape drawing subroutine would use the dot plotting subroutine to produce the shape. This is typical of the way that assembly language subroutines that do complicated things usually rely on simpler assembly language programs to do the 'dirty work' for them. Another advantage of using subroutines as the building blocks of larger programs is that, in principle, any errors in the program should be isolated within a subroutine and so any changes that are necessary to put the error right should be isolated to a small area of the program. However, this neat theory of isolated errors will only work if you follow a strict method of writing subroutines so that they don't interact in ways that you never intended.

Surprisingly the same problem of unwanted subroutine interaction occurs in BASIC. For example, if a subroutine uses a variable with the same name as a variable used by another subroutine then the using either of the subroutines is likely to alter what the other one does. In the same way it is important that assembly language subroutines don't alter values in memory and in registers that other subroutines are relying on to stay the same. This can be achieved by all subroutines using memory locations to store any input values to the subroutine, any output values that are produced, and any temporary or permanent variables created in the process. This is the technique used in the squash program in Chapter Eight. The use of RAM storage for all data allows any subroutine to use all of the registers for its internal calculations. However, there is an alternative approach which involves requiring all subroutines to not change values stored in any register. This sounds like an impossible requirement. How can a subroutine work without changing any of the registers? The answer is that the first instruction of the subroutine pushes all of the registers onto the system stack and the last instruction pulls them all off again. In this way every subroutine saves the register values on entry and restores them after use. For example, a delay subroutine using the first method could be written as follows:

```
@DELAY   LDX    @TIME
@DLOOP   LEAX   -1,X
         BNE    @DLOOP
         RTS
@TIME    FDB    0
```

where the memory location @TIME is used to specify the desired delay before calling the subroutine. Notice that the X register is freely used by this subroutine and there is no attempt to restore its original value at the end of the subroutine. The second method of writing the subroutine would give:

```
@DELAY   PSHS    CC,X
         LDX     @TIME
@DLOOP   LEAX    -1,X
         BNE     @DLOOP
         PULS    CC,X
         RTS
@TIME    FDB     0
```

The only difference is that the CC and X register are pushed onto the system stack and restored at the end of the subroutine. As the X and the CC register are the only two registers used by the subroutine we can guarantee that the subroutine produces its effect, i.e. a delay, without changing the values stored in any register.

Whichever method you use to isolate undesirable side effects of one subroutine on another it is important that you stick to it because mixing the two approaches is certain to cause more confusion than no method at all!

Once you have successfully isolated subroutines there remains the problem of how to pass data to and from subroutines. The simplest method is the one that has been used in all the previous examples i.e. the use of memory locations following the subroutine. However this does have a number of drawbacks when it comes to more advanced applications where machine code subroutines are being used by assembly language programs. In this case it may be difficult to discover the address of the memory location used to store the data. One solution is to use the system stack to pass information in a way that doesn't involve knowing a address. For example, the delay subroutine could pull the value of the X register from the system stack instead of using a fixed memory location. However, you have to be careful to remember that the return address is stored on the system stack after a JSR or BSR instruction.

```
@DELAY   LDX     -2,S
@DLOOP   LEAX    -1,X
         BNE     @DLOOP
         RTS
```

The @DELAY subroutine now requires the delay value to be pushed onto the stack before it is called

```
LDY      #$1000
PSHS     Y
BSR      @DELAY
LEAS     -2,S
```

Notice that after pushing the value $1000 on the stack the BSR instruction automatically pushes the return address in front of the value. This is the reason that the @DELAY subroutine has to take its data from -2,S rather than 0,S. Also notice that after using the subroutine the stack has to be cleaned up either by pulling a two byte value off the stack or, as shown above, simply by subtracting 2 from the S register. The topic of using the stack for temporary storage in conjunction with subroutines is too large a subject to be pursued any further here. (For further information see "The 6809 Companion" by Mike James, published by Babani, 1982.)

The role of BASIC in assembler

The previous section concerning the use of subroutines completely ignores one of the most important sources of machine code subroutines - the BASIC ROM. The BASIC ROM contains subroutines that are used to implement everything that you can do in BASIC. In other words, it contains subroutines to make sounds, read the joystick inputs, plot lines do arithmetic and so on. The only problem is that there isn't a complete list describing what subroutines are to be found where, and, perhaps worse, there is no guarantee that any of the subroutines will remain in the same place in future versions of the Dragon's ROM. The details of some of the subroutines that are officially given by Dragon Data can be found in Appendix IV. It is fairly safe to use these subroutines but any others may carry the risk that your software won't work on future Dragons. To find out about other subroutines there is no alternative but to use a disassembler (see later) and fathom out the way that the BASIC ROM works for yourself.

BASIC can also be useful to the assembly language program in another way. It is often worth writing a program or at least part of a program in BASIC to check that your ideas work before converting it to assembler. You will find many examples of BASIC programs doing jobs that are better suited to assembly language in "The Anatomy of the Dragon".

As well as being useful for checking that a programming idea works before committing yourself to assembly language, there is no avoiding the fact that BASIC is better than assembler for some tasks. In Chapter Nine the one of the sound producing programs was written to be used in combination with a BASIC program that provided it with data by prompting the user in a way that would have been very tedious to write in assembler. Where speed isn't required then BASIC is generally to be preferred. For this reason the ideal way to write a program is to use a mixture BASIC and assembler.

As well as the method of using BASIC with assembler illustrated in Chapter Nine (involving the CLOADM and EXEC instructions) there is also the USR function method. The Dragon's machine code USR function is best reserved for occasions when an assembly language program needs to be used as part of a calculation - and this is not as often as you might expect. For example, in the unlikely event that you had invented a faster way of working out a square root you might write an assembly language program to do just this. In this case the correct way to use it from BASIC would be via the USR function. A machine code USR function must first be defined by -

```
DEFUSRn = XXXX
```

where n is a number in the range 0 to 9 used to identify 1 of 10 possible machine code functions and XXXX is the address of the start of the machine code in question. Once a USR function has been defined it can be called in the same way that any other function can - simply by using its name. If the improved square root machine code program started at $7000 then following -

```
DEFUSR0 = &H7000
```

you could use the function thus -

```
A=USR00(4)
```

(Notice that the DEFUSR0 identifies the function using only a single digit but the function itself uses two digits, i.e. USR00 rather than USR0.) This function would find the square root of four and place the result in the BASIC variable A. You can use a machine code function in exactly the same way that you use an ordinary function. For example,

```
A=2*USR00(X*4+2)+4
```

is perfectly valid. This adequately describes how a machine code USR function can be called from BASIC but it says nothing about how the machine code function itself gets its input value or what it should do with its result to ensure that it is stored in the variable A.

When BASIC transfers control to a machine code USR function it sets the X register to point to an area of memory known as the 'floating point accumulator' or 'FAC'. This holds the result of evaluating the expression contained within the brackets when the function was called. The only trouble is that this value is stored in a representation that would take too long to explain in any detail here called 'floating point binary'. The essence of this representation is that a number is stored in five bytes, the first of which (the one that the X register points at) being the binary exponent + 128. The next four bytes i.e. X + 1, X + 2, X + 3 and X + 4 contain the mantissa in normalised form. In the FAC a sixth byte is also stored at X + 5 which is a copy of the most significant byte of the mantissa but with b7 set to 1 if the number is negative and b7 set to 0 if the number is positive. For a machine code function to do anything useful with this floating point number would be rather complicated so it is usual to convert it to a 16-bit integer using the machine code subroutine stored at $8B2D which returns the value of the FAC in the D register. To convert a value in the D register back into floating point form use the subroutine at $8C37 which will leave the value in the FAC. As the result that is returned by the USR function is the value that is stored in the FAC before the final RTS returns control to BASIC, subroutine $8C37 can be used to store the correct floating point value when the USR function has finished its calculations.

As an example, consider how a simple subroutine to multiply a 16-bit number by two could be implemented as a USR function -

```
@FACINT   EQU     $8B2D
@INTFAC   EQU     $8C37
@TWO      JSR     @FACINT
          ASLB
          ROLA
          JSR     @INTFAC
          RTS
```

(This subroutine multiplies by 2, by doing a single left shift on the A and B registers.) Once you have assembled this subroutine you can test it by typing in -

```
1 DEFUSR0 = &H7000
2 INPUT A
3 PRINT USR00(A)
4 GOTO2
```

You could try using the function in even more complicated expressions just to check that everything works as predicted.

If you want to access any variables, strings or arrays using a machine code USR function then it is worth knowing that the VARPTR function will leave the address of any variable in the FAC. For example, USR01(VARPTR("A$")) leaves the address of the first byte of the string description in the FAC (see "The Anatomy of the Dragon").

Assemblers and other packages

Now that you have worked your way though this brief introductory look at Dragon assembler you must tackle some assembly language projects to develop your skills yet further. Before you can do this, however, it is worth investing in a good quality assembler. Two commercial assemblers are described in Appendix II and now that you have experience of how an assembler works via the BASIC assembler you should be in a position to evaluate such software for yourself! However, if buying a ready made assembler seems like the easy way out then you might like instead to tackle the large project of writing your own assembler (in assembler!) taking the BASIC assembler as your model. In case you choose this course, and for general interest, it is worth pointing out some of the features that have been

left out of the BASIC assembler that are commonly found in commercial assemblers.

The idea of a label to represent an address is usually taken one stage further to include 'address' or 'label expressions'. For example, if @TABLE is a label that marks the start of a sequence of memory locations that are being used to store data, then it is often the case that a particular instruction always needs to make reference to say the second location in the table. Using a label expression this would be written as @TABLE+ 1, in other words, the memory location whose address is given by adding one to the address value that corresponds to @TABLE. Notice that this expression, i.e. @TABLE+ 1, is evaluated by the assembler and used as the address in an instruction. In contrast the seemingly similar

```
LDX     #@TABLE
LEAX    1,X
```

causes the 6809 to work out the same address only when the machine code program is finally run. It is this potential for confusion that makes it wise to leave address expressions alone until you are completely happy with the simpler aspects of addressing.

Commercial assemblers also offer a wider range of constants than just the hex and decimal numbers that the BASIC assembler will handle. Most will allow you to specify numbers in binary and will automatically convert characters to their corresponding ASCII codes automatically. For example, in the DASM assembler -

```
LDA #!A
```

will load the A register with the ASCII code for "A" into the A register. Also all commercial assemblers will automatically convert negative numbers used in address expressions into the correct two's complement form. For example,

```
LDA #-1
```

will load the A register with the two's complement representation of -1 i.e. $FF. This facility is not available not in the BASIC assembler as it stands.

All of these features and more could be added to the BASIC assembler but it would still not be up to the job of assembling large programs because of the time it takes to assemble each line.

As well as a good efficient assembler you also need some way of debugging a machine code program. The most usual way is to use a machine code monitor program such as DEMON from Compusense or DREAMBUG which is contained on the ALLDREAM cartridge from Dragon Data. A monitor is a program that allows you to examine areas of memory, discover the contents of registers and trace the execution of a program instruction by instruction. Whenever a machine code program fails to work, it is either a matter of some very obvious mistake needing to be put right or the machine goes quiet and you are left to ponder what might have gone wrong! In this second situation a machine code monitor is the only way that you can check the misbehaving program instruction by instruction to test that it does what you expect it to.

Another very useful program for any assembly language programmer is a disassembler. This is in essence an assembler working backwards! It takes the codes stored in memory and translates them back to the assembly language mnemonics that represent them. In principle, using a disassembler on a chunk of machine code will produce an assembly language program that can be understood. In practice, trying to understand any assembly language program that you didn't write is very difficult and the output from a dissasembler doesn't include any friendly address labels indicating what the code is used for or what any data locations hold. However, disassembling your machine's BASIC ROM is one very good way of obtaining examples of assembly language programming to study. Many an education in assembly language has come about because of the need to disassemble BASIC ROM!

Appendix I

Instruction Codes

Instruction	Addressing Modes					5	3	2	1	0
Forms	Imm	Dir	Ind	Ext	Inh	H	N	Z	V	C
ABX					3A	—	—	—	—	—
ADCA	89	99	A9	B9		a	a	a	a	a
ADCB	C9	D9	E9	F9		a	a	a	a	a
ADDA	8B	9B	AB	BB		a	a	a	a	a
ADDB	CB	DB	EB	FB		a	a	a	a	a
ADDD	C3	D3	E3	F3		—	a	a	a	a
ANDA	84	94	A4	B4		—	a	a	0	—
ANDB	C4	D4	E4	F4		—	a	a	0	—
ANDCC	1C					*				
ASLA					48	u	a	a	a	a
ASLB					58	u	a	a	a	a
ASL		08	68	78		u	a	a	a	a
ASRA					47	u	a	a		
ASRB					57		a	a		
ASR		07	67	77			a	a	—	a

BITA	85	95	A5	B5		‒ a a 0 ‒
BITB	C5	D5	E5	F5		‒ a a 0 ‒
CLRA					4F	‒ 0 1 0 0
CLRB					5F	‒ 0 1 0 0
CLR		0F	6F	7F		‒ 0 1 0 0
CMPA	81	91	A1	B1		u a a a a
CMPB	C1	D1	E1	F1		u a a a a
CMPD	1083	1093	10A3	10B3		‒ a a a a
CMPS	118C	119C	11AC	11BC		‒ a a a a
CMPU	1183	1193	11A3	11B3		‒ a a a a
CMPX	8C	9C	AC	BC		‒ a a a a
CMPY	108C	109C	10AC	10BC		‒‒ a a a a
COMA					43	‒ a a 0 1
COMB					53	‒ a a 0 1
COM		03	63	73		‒ a a 0 1
CWAI	3C					*
DAA					19	‒ a a 0 a
DECA					4A	‒ a a a ‒‒
DECB					5A	‒ a a a ‒‒
DEC		0A	6A	7A		‒ a a a ‒‒
EORA	88	98	A8	B8		‒ a a 0 ‒‒
EORB	C8	D8	E8	F8		‒ a a 0
EXG	1E					‒ ‒ ‒ ‒ ‒
INCA					4C	‒ a a a ‒‒
INCB					5C	‒ a a a ‒‒
INC		0C	6C	7C		‒ a a a ‒‒
JMP		0E	6E	7E		‒ ‒ ‒ ‒ ‒
JSR		9D	AD	BD		‒ ‒ ‒ ‒ ‒
LDA	86	96	A6	B6		‒ a a 0 ‒

LDB	C6	D6	E6	F6		‒ a a 0 ‒
LDD	CC	DC	EC	FC		‒ a a 0 ‒
LDS	10CE	10DE	10EE	10FE		‒ a a 0 ‒
LDU	CE	DE	EE	FE		‒ a a 0 ‒
LDX	8E	9E	AE	BE		‒ a a 0 ‒
LDY	108E	109E	10AE	10BE		‒ a a 0 ‒
LEAS			32			‒ ‒ ‒ ‒ ‒
LEAU			33			‒ ‒ ‒ ‒ ‒
LEAX			30			‒ ‒ a ‒ ‒
LEAY			31			‒ ‒ a ‒ ‒
LSLA					48	‒ a a a a
LSLB					58	‒ a a a a
LSL		08	68	78		‒ a a a a
LSRA					44	‒ 0 a ‒ a
LSRB					54	‒ 0 a ‒ a
LSR		04	64	74		‒ 0 a ‒ a
MUL					3D	‒ ‒ a ‒ s
NEGA					40	u a a a a
NGB					50	u a a a a
NEG		00	60	70		u a a a a
NOP					12	‒ ‒ ‒ ‒ ‒
ORA	8A	9A	AA	BA		‒ a a 0 ‒
ORB	CA	DA	EA	FA		‒ a a 0 ‒
ORCC	1A					*
PSHS	34					‒ ‒ ‒ ‒ ‒
PSHU	36					‒ ‒ ‒ ‒ ‒
PULS	35					‒ ‒ ‒ ‒ ‒
PULU	37					‒ ‒ ‒ ‒ ‒
ROLA					49	‒ a a a a
ROLB					59	‒ a a a a

ROL		09	69	79		_ a a a a
RORA					46	_ a a _ a
RORB					56	_ a a _ a
ROR		06	66	76		_ a a _ a
RTI					3B	*
RTS					39	_ _ _ _ _
SBCA	82	92	A2	B2		u a a a a
SBCB	C2	D2	E2	F2		u a a a a
SEX					1D	_ a a 0 _
STA		97	A7	B7		_ a a 0 _
STB		D7	E7	F7		_ a a 0 _
STD		DD	ED	FD		_ a a 0 _
STS		10DF	10EF	10FF		_ a a 0 _
STU		DF	EF	FF		_ a a 0 _
STX		9F	AF	BF		_ a a 0 _
STY		109F	10AF	10BF		_ a a 0 _
SUBA	80	90	A0	B0		u a a a a
SUBB	C0	D0	E0	F0		u a a a a
SUBD	83	93	A3	B3		_ a a a a
SWI					3F	_ _ _ _ _
SWI2					103F	_ _ _ _ _
SWI3					113F	_ _ _ _ _
SYNC					13	_ _ _ _ _
TFR	1F					_ _ _ _ _
TSTA					4D	_ a a 0 _
TSTB					5D	_ a a 0 _
TST		0D	6D	7D		_ a a 0 _

Key to Condition Codes
__ not affected

a affected - test and set if true, clear otherwise
u value of half carry flag is undefined
s special case - carry set if b7 is set
* condition codes set as a direct result of the instruction

Branch Instructions

Form	Rel	Long	Rel
BCC	24	LBCC	1024
BCS	25	LBCS	1025
BEQ	27	LBEQ	1027
BGE	2C	LBGE	102C
BGT	2E	LBGT	102E
BHI	22	LBHI	1022
BHS	24	LBHS	1024
BLE	2F	LBLE	102F
BLO	25	LBLO	1025
BLS	23	LBLS	1023
BLT	2D	LBLT	102D
BMI	2B	LBMI	102B
BNE	26	LBNE	1026
BPL	2A	LBPL	102A
BRA	20	LBRA	16
BRN	21	LBRN	1021
BSR	8D	LBSR	17
BVC	28	LBVC	1028
BVS	29	LBVS	1029

Indexed Addressing Modes

Non-Indirect

Type	Forms	Assembler Form	Post Byte OP Code
Constant Off-set	Zero offset	,R	1RR00100
	5-bit offset	n,R	0RRnnnnn
	8-bia offset	n,R	1RR01000
	16-bit offset	n,R	1RR01001
Accumulator Offet	A register offset	A,R	1RR00110
	B register offset	B,R	1RR00101
	D register offset	D,R	1RR01011
Auto Increment/ Auto Decrement	Increment by 1	,R+	1RR00000
	Increment by 2	,R++	1RR00001
	Decrement by 1	,-R	1RR00010
	Decrement by 2	,--R	1RR00011
PC Relative	8-bit offset	n,PCR	1XX01100
	16-bit offset	n,PCR	1XX01101

Indirect

Type	Forms	Assembler Form	Post Byte OP Code
Constant Offset	Zero offest	[,R]	1RR10100
	5-bit offset	defaults to 8-bit	
	8-bit offset	[n,R]	1RR11000
	16-bit offset	[n,R]	1RR11001

Accumulator Offet	A register offset	(A,R)	1RR10110
	B register offset	(B,R)	1RR10101
	D register offset	(D,R)	1RR11011
Auto Increment/	Increment by 1	not allowed	
Auto Decrement	Increment by 2	(,R+ +)	1RR10001
	Decrement by 1	not allowed	
	Decrement by 2	(,--R)	1RR10011
PC Relatiue	8-bit offset	(n,PCR)	1XX11100
	16-bit offset	(n,PCR)	1XX11101
Extended Indirect	16-bit address	(n)	10011111

Key
R = register code
X = don't care

Register Code X = 00 Y = 01 U = 10 S = 11

Appendix II

Two Commercial Assemblers

DASM

The DASM assembler from Compusense has a number of features that make it particularly suitable for the beginning assembly language programmer. Available as a program cartridge it will assemble lines of 6809 assembly language embedded in a Dragon BASIC program. The machine code so produced can be stored anywhere in memory but it is usual for DASM to store it just above the memory used by BASIC (as with the BASIC assembler). Since the text of the assembly language program is entered exactly as if it formed part of a BASIC program, editing the text is carried out, by the familiar but limited, EDIT command. DASM supports all of the 6809's features including indirect addressing, Program Counter relative etc. It also supports full address expressions, a range of constant types and the usual pseudo ops. All labels in DASM must start with @ and so it is possible to assembly and run any of the programs in this book without modification.

If you want to use a machine code monitor program to debug your programs assembled with DASM, then Compusense have produced a simple monitor called DEMON, available as a separate cartridge or together with DASM on a single cartridge. This is a fairly limited but easy-to-use debugging aid. It includes a memory dump, a register examine facility and break points but not a disassembler.

In conclusion, DASM is easy to use an especially suited for situations in which a little assembly language has to be mixed with BASIC.

DREAM

Dragon Data's own assembler is well described by its name. For the serious assembly language programmer it is indeed a dream come true! The

only problem is that it takes over the entire machine and substitutes its own editor in place of BASICs crude but familiar editing commands. If you are prepared to learn to use the DREAM editor then you will very quickly find it so useful that you will be using it for other editing tasks! DREAM is available either as a cassette or a cartridge and there is also a cartridge version called ALLDREAM. This last package is by far the best investment if you are planning to do much assembly language programming as it frees the largest amount of RAM for use and also includes gives direct acess to the DREAMBUG monitor program. This monitor is very sophisticated and as well as the standard features such as memory and register examine, break points etc it also contains a trace facility and a disassembler.

The DREAM assembler supports all the 6809's instructions and addressing modes. It also supports the full range of address expressions and constants. It also uses the most economical representation for constant offset indexed mode even if this means making more than two passes through the program. Only one label in any program to be assembled by DREAM can start with @ which is used as an optional marker for the start of the program. In other words, to use DREAM to assemble the programs in this book REMOVE THE @ SYMBOL FROM EVERY LABEL. Apart from this slight change you should have no trouble using the DREAM assembler.

In conclusion, the DREAM assembler and the DREAMBUG monitor form an ideal pair for anyone planning to do assembly language programming on a regular basis.

Appendix III

Complete Listing of Assembler

```
 1 REMBASICASSEMBLERV10.1

 5 CLEAR2000,&H6FFF

10 DATALDA,&H86,&H96,&HA6,&HB6,-1
11 DATALDB,&HC6,&HD6,&HE6,&HF6,-1
12 DATASTA,-1,&H97,&HA7,&HB7,-1
13 DATASTB,-1,&HD7,&HE7,&HF7,-1
14 DATAADDA,&H8B,&H9B,&HAB,&HBB,-1
15 DATAADDB,&HCB,&HDB,&HEB,&HFB,-1
16 DATARTS,-1,-1,-1,-1,&H39
17 DATAJMP,-1,&H0E,&H6E,&H7E,-1
18 DATAJSR,-1,&H9D,&HAD,&HBD,-1
19 DATAANDA,&H84,&H94,&HA4,&HBB,-1
20 DATAANDB,&HC4,&HD4,&HE4,&HF4,-1
21 DATAORA,&H8A,&H9A,&HAA,&HBA,-1
22 DATAORB,&HCA,&HDA,&HEA,&HFA,-1
23 DATAEORA,&H88,&H98,&HA8,&HB8,-1
24 DATAEORB,&HC8,&HD8,&HE8,&HF8,-1
25 DATACOMA,-1,-1,-1,-1,&H43
26 DATACOMB,-1,-1,-1,-1,&H53
27 DATACOM,-1,&H03,&H63,&H73,-1
28 DATALSLA,-1,-1,-1,-1,&H48
29 DATALSLB,-1,-1,-1,-1,&H58
30 DATALSL,-1,&H08,&H68,&H78,-1
31 DATALSRA,-1,-1,-1,-1,&H44
32 DATALSRB,-1,-1,-1,-1,&H54
```

```
33 DATA LSR,-1,&H04,&H64,&H74,-1
34 DATA ROLA,-1,-1,-1,-1,&H49
35 DATA ROLB,-1,-1,-1,-1,&H59
36 DATA ROL,-1,&H09,&H69,&H79,-1
37 DATA RORA,-1,-1,-1,-1,&H46
38 DATA RORB,-1,-1,-1,-1,&H56
39 DATA ROR,-1,&H06,&H66,&H76,-1
40 DATA ADDD,&HC3,&HD3,&HE3,&HF3,-1
41 DATA SUBA,&H80,&H90,&HA0,&HB0,-1
42 DATA SUBB,&HC0,&HD0,&HE0,&HF0,-1
43 DATA SUBD,&H83,&H93,&HA3,&HB3,-1
44 DATA CLRA,-1,-1,-1,-1,&H4F
45 DATA CLRB,-1,-1,-1,-1,&H5F
46 DATA CLR,-1,&H0F,&H6F,&H7F,-1
47 DATA INCA,-1,-1,-1,-1,&H4C
48 DATA INCB,-1,-1,-1,-1,&H5C
49 DATA INC,-1,&H0C,&H6C,&H7C,-1
50 DATA DECA,-1,-1,-1,-1,&H4A
51 DATA DECB,-1,-1,-1,-1,&H5A
52 DATA DEC,-1,&H0A,&H6A,&H7A,-1
53 DATA NEGA,-1,-1,-1,-1,&H40
54 DATA NEGB,-1,-1,-1,-1,&H50
55 DATA NEG,-1,&H00,&H60,&H70,-1
56 DATA STD,-1,&HDD,&HED,&HFD,-1
57 DATA LDD,&HCC,&HDC,&HEC,&HFC,-1
58 DATA SEX,-1,-1,-1,-1,&H1D
59 DATA ASRA,-1,-1,-1,-1,&H47
60 DATA ASRB,-1,-1,-1,-1,&H57
61 DATA ASR,-1,&H07,&H67,&H77,-1
62 DATA ASLA,-1,-1,-1,-1,&H48
63 DATA ASLB,-1,-1,-1,-1,&H58
64 DATA ASL,-1,&H08,&H68,&H78,-1
65 DATA MUL,-1,-1,-1,-1,&H3D
66 DATA DAA,-1,-1,-1,-1,&H19
67 DATA CMPA,&H81,&H91,&HA1,&HB1,-1
68 DATA CMPB,&HC1,&HD1,&HE1,&HF1,-1
69 DATA CMPD,&H1083,&H1093,&H10A3,&H10B3,-1
70 DATA BITA,&H85,&H95,&HA5,&HB5,-1
71 DATA BITB,&HC5,&HD5,&HE5,&HF5,-1
72 DATA ANDCC,&H1C,-1,-1,-1,-1
73 DATA ORCC,&H1A,-1,-1,-1
```

```
 74 DATATSTA,-1,-1,-1,-1,&H4D
 75 DATATSTB,-1,-1,-1,-1,&H5D
 76 DATATST,-1,&H0D,&H6D,&H7D,-1
 77 DATAADCA,&H89,&H99,&HA9,&HB9,-1
 78 DATAADCB,&HC9,&HD9,&HE9,&HF9,-1
 79 DATASBCA,&H82,&H92,&HA2,&HB2,-1
 80 DATASBCB,&HC2,&HD2,&HE2,&HF2,-1
 81 DATALDX,&H8E,&H9E,&HAE,&HBE,-1
 82 DATALDY,&H108E,&H109E,&H10AE,&H10BE,-1
 83 DATALDS,&H10CE,&H10DE,&H10EE,&H10FE,-1
 84 DATALDU,&HCE,&HDE,&HEE,&HFE,-1
 85 DATASTS,-1,&H10DF,&H10EF,&H10FF,-1
 86 DATASTU,-1,&HDF,&HEF,&HFF,-1
 87 DATASTX,-1,&H9F,&HAF,&HBF,-1
 88 DATASTY,-1,&H109F,&H10AF,&H10BF,-1
 89 DATACMPS,&H118C,&H119C,&H11AC,&H11BC,-1
 90 DATACMPU,&H1183,&H1193,&H11A3,&H11B3,-1
 91 DATACMPX,&H8C,&H9C,&HAC,&HBC,-1
 92 DATACMPY,&H108C,&H109C,&H10AC,&H10BC,-1
 93 DATALEAS,-1,-1,&H32,-1,-1
 94  ATALEAU,-1,-1,&H33,-1,-1
 95 DATALEAX,-1,-1,&H30,-1,-1
 96 DATALEAY,-1,-1,&H31,-1,-1
 97 DATAABX,-1,-1,-1,-1,&H3A
 98 DATAEXG,&H1E,-1,-1,-1,-1
 99 DATATFR,&H1F,-1,-1,-1,-1
100 DATAPULS,&H35,-1,-1,-1,-1
101 DATAPULU,&H37,-1,-1,-1,-1
102 DATAPSHS,&H34,-1,-1,-1,-1
103 DATAPSHU,&H36,-1,-1,-1,-1
104 DATARTI,-1,-1,-1,-1,&H3B
105 DATACWAIT,&H3C,-1,-1,-1,-1
106 DATASYNC,-1,-1,-1,-1,&H13
107 DATASWI,-1,-1,-1,-1,&H3F
108 DATASWI2,-1,-1,-1,-1,&H103F
109 DATASWI3,-1,-1,-1,-1,&H113F
199 DATAZZZ,-1,-1,-1,-1,-1

400 DATABRA,&H20,&H16
401 DATABCC,&H24,&H1024
402 DATABCS,&H25,&H1025
```

```
403 DATABEQ,&H27,&H1027
404 DATABGE,&H2C,&H102C
405 DATABGT,&H2E,&H102E
406 DATABHI,&H22,&H1022
407 DATABHS,&H24,&H1024
408 DATABLE,&H2F,&H102F
409 DATABLO,&H25,&H1025
410 DATABLS,&H23,&H1023
411 DATABLT,&H2D,&H102D
412 DATABMI,&H2B,&H102B
413 DATABNE,&H26,&H1026
414 DATABPL,&H2A,&H102A
415 DATABSR,&H8D,&H17
416 DATABVC,&H28,&H1028
417 DATABVS,&H29,&H1029
499 DATAZZZ,-1,-1


500 GOSUB1000
510 GOSUB2000
515 FORPASS=1TO2
518 I=1:P=&H7000
520 GOSUB3000
530 GOSUB4000
540 GOSUB5000
550 IFPS=0THENGOSUB6000
555 IFPS>0THENGOSUB6500
560 I=I+1:PS=0
570 IFI<=TTHENGOTO520
575 PRINT
580 NEXTPASS
590 I=T:GOSUB1980
600 LC=0:GOTO515


1000 DIMA$(150),C(5),T$(50),T(50)
1010 I=0
1020 P=&H7000
1030 LC=0
1040 RETURN
```

```
1980 PRINT"PRESS ANY KEY TO CONTINUE";
1990 IF INKEY$ = "" THEN GOTO 1990
2000 CLS
2010 PRINT@66,"B A S I C  A S S E M B L E R"
2020 PRINT
2030 PRINTTAB(10); "SELECT ONE OF"
2040 PRINT
2050 PRINTTAB(8); "INPUT/EDIT ..... 1"
2060 PRINTTAB(8); "ASSEMBLE.....2"
2070 PRINTTAB(8); "SAVE ON TAPE....3"
2080 PRINTTAB(8); "LOAD FROM TAPE..4"
2090 PRINTTAB(8); "EXEC PROGRAM....5"
2100 INPUT ACTION
2110 IF ACTION< 1 OR ACTION> 5 THEN GOTO 2000
2120 ON ACTION GOTO 2200,2800,2850,2920,2990

2200 CLS
2210 PRINT@76, "E D I T"
2220 PRINT
2230 PRINTTAB(10);"SELECT ONE OF"
2240 PRINT
2250 PRINTTAB(8);"LIST PROGRAM....1"
2260 PRINTTAB(8);"LIST TO PRINTER.2"
2270 PRINTTAB(8);"ADD TO PROGRAM..3"
2280 PRINTTAB(8);"DELETE LINES....4"
2290 INPUT ED
2300 IF ED< 1 OR ED> 4 THEN GOTO 2000
2310 ON ED GOTO 2400,2400,2500,2700

2400 CLS
2405 IF I= 0 THEN GOTO 1980
2410 FOR K = 1 TO I
2420 IF ED= 1 THEN PRINT K;":";TAB(4);A$(K) ELSE
PRINT # 2,K;":";TAB(4);A$(K)
2430 NEXT K
2440 GOTO 1980

2500 IF I= 0 THEN GOTO 2620
2505 INPUT"ADD LINES FOLLOWING LINE NUMBER";LN
2510 IF LN> = I THEN LN = I:GOTO 2620
2520 INPUT "NUMBER OF LINES TO INSERT";IN
```

```
2530 IFI+IN>150 THEN PRINT"TOO MANY":GOTO2000
2540 FORK=ITOLN+1 STEP-1
2550 A$(K+IN)=A$(K)
2560 NEXTK
2570 FORK=LN+1TOLN+IN
2580 PRINTK;":";TAB(4);
2590 LINE INPUT A$(K)
2600 NEXT K
2605 I=I+IN
2610 GOTO1980
2620 PRINT"TYPEENDTOFINISH"
2630 K=I+1
2640 PRINTK;":";TAB(4);
2650 LINEINPUTL$
2660 IFLEFT$(L$,3)="END" THENGOTO1980
2670 I=K:A$(I)=L$
2680 GOTO2630


2700 INPUT"FIRSTLINETODELETE";FL
2710 INPUT"LASTLINETODELETE";LL
2720 IFLL<FLTHENPRINT"NOTDELETED":GOTO1980
2730 FORK=LL+1TOI
2740 A$(FL+K-LL-1)=A$(K)
2750 NEXTK
2760 I=I-(LL-FL+1):PRINT"DELETED"
2770 GOTO1980
2800 INPUT"SCREEN(0)ORPRINTER(1)";PRT
2810 T=I
2820 PRT=PRT*2
2830 RETURN


2850 INPUT"FILENAME";F$
2860 PRINT"PRESSPLAYANDRECORD"
2870 PRINT"PRESSANYKEYWHENREADY"
2880 IFINKEY$=""THENGOTO2880
2890 OPEN"O",#-1,F$
2900 FORK=1TOI:PRINT#-1,A$(K):NEXTK
2910 CLOSE#-1:GOTO1980
```

```
2920 INPUT"FILENAME";F$
2930 PRINT"PRESSPLAY"
2940 OPEN"I",#-1,F$
2950I=0
2960 IFEOF(-1)THENCLOSE#-1:GOTO1980
2970 I=I+1:INPUT#-1,A$(I)
2980 GOTO2960

2990 CLS:EXEC&H7000
2995 GOTO1980

3000 J=1
3010 IFMID$(A$(I),j,1)=""THENJ=J+1:GOTO3010
3020 M$=MID$(A$(I),J,1)
3030 J=J+1
3040 IF J< =LEN(A$(I)) THEN IF MID$(A$(I),J,1)< > " " THEN
M$=M$+MID$(A$(I),J,1):J=J+1:GOTO3040
3050 J=J+1
3060 IFLEFT$(M$,1)="@"THENGOTO3500
3070 RETURN
3500 S$=M$
3510 GOSUB7000
3520 IFF>0ANDPASS=1THENERR=2:GOTO9000
3525 IFF>0ANDPASS=2THENGOTO3010
3530 LC=LC+1
3540 T$(LC)=M$
3550 T(LC)=P
3560 GOTO3010

4000 RESTORE
4001 IFM$="EQU"THENPS=1:RETURN
4002 IFM$="RMB"THENPS=2:RETURN
4003 IFM$="FCB"THENPS=3:RETURN
4004 IFM$="FDB"THENPS=4:RETURN
4009 IF(LEFT$(M$,1)="B"ANDLEFT$(M$,3)< > "BIT")OR
LEFT$(M$,2)="LB"THENGOTO4500
4010READC$
4015 FORK=1TO5:READC(K):NEXTK
4020 IFC$="ZZZ"THENI=I+1:ER=1:GOTO9000
4030  FC$=M$THENRETURN
4040 GOTO4010
```

```
4500 READC$
4510 FORK=1TO5:READC(K):NEXTK
4520 IFC$<>"ZZZ"THENGOTO4500
4530 IF LEFT$(M$,1)="L" THEN M$=RIGHT$(M$,3): BR=2 ELSE
BR=14535TYPE=BR
4540 READC$
4550 FORK=1TO2:READC(K):NEXTK
4560 IFC$="ZZZ"THENI=I+1:ER=1:GOTO9000
4570 IFC$=M$THENRETURN
4580 GOTO4540

5000 IFM$="EXG"ORM$="TFR"THENGOSUB5100
5005 IF LEFT$(M$,3)="PUL" OR LEFT$(M$,3)="PSH" THEN
GOSUB5300ELSEGOSUB5500
5010 IFAF$=""THENTYPE=5:RETURN
5020 A=VAL(AF$)
5024 IFBR>0THENGOTO5700
5025 IFTYPE=1THENRETURN
5030 IFTYPE=2THENRETURN
5035 IFTYPE=3THENRETURN
5040 TYPE=4
5050 RETURN

5100 FORK=JTOLEN(A$(I))
5110 IFMID$(A$(I),K,1)=","THENGOTO5130
5120 NEXTK
5130 L$=MID$(A$(I),K-2,2)
5140 GOSUB5200
5150 AF$=L$
5160 L$=MID$(A$(I),K+1,2)
5165 IF RIGHT$(L$,1)=" " OR LEN(L$)=1 THEN L$="
"+LEFT$(L$,1)
5170 GOSUB5200
5180 AF$="&H"+AF$+L$
5185 TYPE=1
190 RETURN

5200 IFL$="D"THENL$="0"
5210 IFL$="X"THENL$="1"
5220 IFL$="Y"THENL$="2"
```

```
5230 IFL$ = "U"THENL$ = "3"
5240 IFL$= "S"THENL$ = "4"
5250 IFL$ = "PC"THENL$ = "5"
5260 IFL$= "A"THENL$ = "8"
5270 IFL$ = "B"THENL$= "9"
5280 IFL$ = "CC"THENL$ = "A"
5290 IF L$ = "DP" THEN L$ = "B"

5295 RETURN
5300 A = 0
5310 FORK = JTOLEN(A$(I))
5320 L$ = MID$(A$(I),K,2)
5330 IFL$ = "PC"THENA = A + 128:K = K + 1
5340 IFL$ = "DP"THENA = A + 8:K = K + 1
5350 IFL$ = "CC"THENA = A + 1:K = K + 1
5360 L$ = MID$(A$(I),K,1)
5370 IFL$= "A"THENA = A + 2
5380 IFL$ = "B"THENA = A + 4
5390 IFL$= "X"THENA = A + 16
5400 IFL$ = "Y"THENA = A + 32
5410 IFL$ = "S"ORL$ = "U"THENA = A + 64
5420 NEXTK
5430 AF$ = STR$(A)
5440 TYPE = 1
5450 RETURN

5500 AF$ = ""
5510 FORK = JTOLEN(A$(I))
5520 L$ = MID$(A$(I),K,1)
5521 IFL$ = ">"THENAF$= "":TYPE = 2:GOTO5540
5522 IFL$ = "#"THENTYPE = 1:AF$ = "":GOTO5540
5250 IFL$ = "$"THENL$ = "&H"
5526 IFL$= ","THENGOTO5800
5530IFL$<>""THENAF$ = AF$ + L$
5540 NEXTK
5550 IF LEFT$(AF$,1)<>"@"THEN RETURN
5560 S$ = AF$
5570 GOSUB7000
5580 IFF = 0 ANDPASS = 2THENERR = 3:GOTO9000
5590 AF$ = STR$(T(F))
5600 RETURN
```

```
5700 IFPASS=1THENA=0:RETURN
5705 OF=A-BR-1-P
5710 IF BR=1 AND (OF<-128 OR OF>127) THEN
ER=4:I=I+1:GOTO9000
5720 IFOF>=0THENA=OF:RETURN
5730 IFBR=2THENGOTO5760
5740 A=256+OF
5750 RETURN
5760 A=65536+OF
5770 RETURN

5800 TYPE=3
5810 L$=MID$(A$(I),K-1,1)
5820 OF=0
5830 IFL$="A"THENOF=&H86
5840 IFL$="B"THENOF=&H85
5850 IFL$="D"THENOF=&H8B
5860 L$=MID$(A$(I),K+1,1)
5870 IFL$="-"THENL$=MID$(A$(I),K+2,1):OF=&H82
5880 IFL$="-"THENL$=MID$(A$(I),K+3,1):OF=&H83
5890 RF=0
5900 IFL$="Y"THENRF=1
5901 IFL$="U"THENRF=2
5902 IFL$="S"THENRF=3
5910 IFMID$(A$(I),K+2,1)="+"THENOF=&H80
5920 IFMID$(A$(I),K+2,2)="++"THENOF=&H81
5930 IFOF<>0THENAF$=STR$(RF+OF):RETURN
5950 OF=&H89+RF
5955 AF$=""+AF$
5960 A=VAL(AF$)
5970 IA=1
5980 IFA>=0THENRETURN
5990 AF$=STR$(65536+A)
5995 RETURN

6000 IFC(TYPE)=-1THENERR=5:GOTO9000
6010 IFPASS=1THENGOTO6200
6020 PRINT#-PRT,RIGHT$(""+HEX$(P),4);TAB(5);
6030 PRINT#-PRT,HEX$(C(TYPE));TAB(8);
6035 IFIA=1THENPRINT#-PRT,HEX$(OF);
```

```
6040 IFTYPE<>5THENPRINT#-PRT,HEX$(A);
6050 PRINT#-PRT,TAB(15);A$(I)
6200 IFC(TYPE)<256THENPOKEP,C(TYPE)
6210 IF C(TYPE)>255 THEN POKE P,INT(C(TYPE)/256): POKE
P+1,C(TYPE)-INT(C(TYPE)/256)*256:P=P+1
6220 P=P+1
6230 IFBR=2THENTYPE=4
6240 IFTYPE=1ANDRIGHT$(M$,1)="D"THENTYPE=4
6241 IFTYPE=1ANDRIGHT$(M$,1)="X"THENTYPE=4
6242 IFTYPE=1ANDRIGHT$(M$,1)="Y"THENTYPE=4
6243 IFTYPE=1ANDRIGHT$(M$,1)="U"THENTYPE=4
6244 IFTYPE=1ANDRIGHT$(M$,1)="S"THENTYPE=4
6250 IFTYPE=5THENTYPE=0:RETURN
6255 IFIA=1THENPOKEP,OF:P=P+1:TYPE=4
6260 IFTYPE=2ORTYPE=1ORTYPE=3THENPOKEP,A
6270 IF TYPE=4 THEN POKE P,INT(A/256):P=P+1: POKE
P,A-INT(A/256)*256
6280 TYPE=0
6290 BR=0
6300 P=P+1
6305 IA=0
6310 RETURN

6500 IFPS<>1THENGOTO6540
6510 IFPASS=1THENT(LC)=A
6520 IFPASS=2THENPRINT#-PRT,TAB(15);A$(I)
6530 RETURN
6540 IFPS<>2THENGOTO6570
6545 IFPASS=2THENPRINT#-PRT,HEX$(P);
6550 P=P+A
6560 IFPASS=2THENGOTO6520
6570 IFPS<>3THENGOTO6650
6580 A=A-INT(A/256)*256
6590 IFPASS=1THENGOTO6620
6600 PRINT#-PRT,HEX$(P);TAB(5);HEX$(A);
6610 PRINT#-PRT,TAB(15);A$(I)
6620 POKEP,A
6630 P=P+1
6640 RETURN
6650 IFPS<>4THENRETURN
6660 IFPASS=1THENGOTO6710
```

```
6670 LB=A-INT(A/256)*256
6680 HB=INT(A/256)
6690 PRINT#-PRT,HEX$(P);TAB(5);HEX$(HB);TAB(8);HEX$(LB);
6700 PRINT#-PRT,TAB(15);A$(I)
6705 POKEP,HB:POKEP+1,LB
6710 P=P+2
6720 RETURN

7000 K=1
7010 IFK>LCTHENF=0:RETURN
7020 IFT$(K)=S$THENF=K:RETURN
7030 K=K+1
7040 GOTO7010

9000 PRINT#-PRT,"ERROR--";ER;"****INLINE";I-1
9010 RETURN
```

Appendix IV

ROM Subroutines

Address	Description
JSR $8015	Turn on cassette relay
JSR $8018	Turn off cassette relay
JSR ($A008)	Write block of data to cassette - $7C = Block type 0 is fileheader 1 is data FF is end of file $7D = Number of bytes to be written $7E/F = Address of start of data to be written
JSR $8021	Prepares cassette for data input
JSR ($A006)	Reads in data from cassette (used following JSR $8021)- $7E/F = Addressoflocationwheredatawillbestored $81 Error code, clear if no error
JSR $8006	Reads keyboard, returns ASCII code of key pressed in A register. If no key is pressed A = 0
JSR $8012	Updates the four joystick reading stored in $15A to $15D
JSR $800C	Writes the character whose ASCII code is in the A registertothescreen- &88/89 contain the address of the next screen location the $800C will use
JSR $800F	As for $800C but character is sent to printer

Answers to Micro Projects

Chapter Two

1)

address	data	
28672	146	LDA 200
28673	200	
28674	153	ADDA201
28675	201	

2) After running the program, memory location 200 still contains 56 and memory location 201 still contains 4. The result of the addition, that is 60, is stored in the A register.

Chapter Three

1) Each pair of hex characters takes single memory location so it takes 4 memory locations to store $F3095E6F.

2)

a) $0100	=	0000	0001	0000	0000
b) $1000	=	0001	0000	0000	0000
c) $7FFF	=	0111	1111	1111	1111
d) $7FFF	=	1111	1111	1111	1111

$FFFF is the highest address that you can use on the Dragon and $7FFF is the highest address occupied by RAM.

3a) Trying to store something using immediate addressing doesn't make any sense.

b) $7FFF is too large to be loaded into the A register as the result of immediate addressing.

4)

```
7000    86      10
7002    BB      7FFF
7005    B7      7FFF
7008    39
```

The program adds $10 to the contents of memory location $7FFF and then stores the result back in $7FFF.

Chapter Four

1)

```
 585 GOSUB 6900
6900 PRINT
6910 FOR K=1 TO LC
6920 PRINT T$(K);" = ";T(k)
6930 NEXT K
6940 RETURN
```

2)

```
@INPUT  EQU     $8006
@PRINT  EQU     $800C
@LOOP   JSR     @INPUT
        JSR     @INPUT
        JMP     @LOOP
```

Chapter Five

1)

```
        LDA     @DATA
        ORA     #$80
        ANDA    #$F1
        STA     @DATA
```

2)

```
@LOOP   COM     @FLIPPER
        JMP     @LOOP
```

3)

```
        LSLA
        ROLB
        LSLA
        ROLB
        LSLA
        ROLB
        LSLA
        ROLB
```

This program is based on the fact that LSLA followed by ROLB will move b7 in the A register into b1 in the B register using the C bit as a temporary store.

4)

```
@START  LDA     @DATA1
        ANDA    @DATA2
        STA     @DATA3
        RTS
@DATA1  FCB     23
@DATA2  FCB     44
@DATA3  FCB     0
```

Notice that the values following the FCB's constitute whatever data you actually wanted to AND together.

Chapter Six

```
        LDA     @DATA1
        ADDA    @DATA2
        STA     @ANS
        RTS
```

This program would be capable of adding together 200 and 50 giving the answer 250 in memory location @ANS.

2) No modifications would be necessary. A two's complement addition program is different only in that you have to interpret the bit patterns that represent the number differently. The program would be capable of adding 105 and −15 leaving the answer 90 in memory location @ANS.

3)

```
LDA     @NUM
ASLA
ASLA
ASLA
ADDA    @NUM
STA     @ANS
```

This program will multiply the two's complement number in @NUM by nine by first performing three arithmetic shift lefts and then adding the original number to the result. As each shift left is equivalent to multiplication by 2, the final result stored in @ANS is 8 times the contents of @NUM plus the contents of @NUM or, in other words, 9 times the contents of @NUM as required.

4)

```
LDB     @LITTLE
SEX
SUBD    @BIG
```

The eight-bit value is loaded into the B register and then sign extended into the A register to give a correct two's complement 16-bit number in the D register. The SUBD instruction is then used to subtract the 16-bit value giving the result, which is also 16 bits, and so takes two memory locations to store.

Chapter Seven

1)

```
@ORCC   $04
ANDCC   $FE
```

2)

```
          LDB 10
@LOOP     .....
          instructions within the loop
          .....
          DECB
          CMPB 1
          BGE       @LOOP
```

3)

```
          LDA       @DATA1
          ADDA      @DATA2
          BCS       @ERJMP
          JSR       @RESULT
          .....
          rest of program
          .....
@ERJMP    JSR       @ERROR
          .....
          rest of program
          .....
```

Following a simple binary addition the carry bit is set if the result is out of range.

4) The only change that is necessary is to change the BCS @ERJMP to BVS @ERJMP. For two's complement addition the V bit is set following an overflow.

Chapter Nine

1)

```
@DELAY    LDX       @TIME
@DLOOP    LEAX      -1
          BNE       @DLOOP
          RTS
```

2)

```
10 CLEAR 1000,&H6FFF
20 POKE &H7EFF,255
30 FOR I=1 TO 255
40 POKE &H7F00+I-1,RND(64)-1
50 NEXT I
60 POKE &H7EFE,RND(255)
70 POKE &H7EFD,RND(255)
80 EXEC &H7000
90 STOP
```

where all of the values of the sound generator program are set randomly.

Chapter Ten

1)

```
@DELAY    SYNC
          RTS
```

Index

About This Book:-

The language of the Dragon computer is BASIC to many people but, to write high speed programs and to get the stunning visual effects that you see in arcade style games, you need to go further and program in Assembler.
To many people, assembler language is a black art, not intended for the average programmer. Mike James shows you that this is just not true and takes you step-by-step through every detail of assembler concepts for the 6809 microprocessor (as used in the Dragon) leading up to those very techniques that you need to write fully professional programs.

And you'll see just what assembler packages to get for your Dragon.
If you've read Mike James' other book - "Anatomy of the Dragon" — you'll be in good shape to read this one. The two books are all you'll need to be a Dragon expert!

Other Books of Interest:-

Anatomy of the Dragon, by M. James
Hot Programs to Feed your Dragon, by G.P.S Robinson and M.A. Smith
The Complete FORTH, by A.F.T. Winfield
Advanced FORTH, by D. Husband
Practical Pascal for Microcomputers, by R. Graham
Practical COBOL for Microcomputers, by K. Sullivan
Programs that write Programs, by C. Naylor
Getting More from your Commodore 64, by M. Harrison
Getting More from your BBC Computer, by N. Kantaris
Getting More from your ORIC, by H. Hicks

About the Author:-

Mike James is a prolific author of books and articles on computing and Microcomputers. This is his second book for Sigma. He now runs Infomax, a computer consultancy.

Published by:-
Sigma Technical Press
5, Alton Road,
Wilmslow, Cheshire.
SK9 5DY, U.K.

ISBN 0 905104 36 6

£6.95

Language of the DRAGON: 6809 ASSEMBLER

Mike James