INTRODUCING DRAGON MACHINE CODE

&H0D &H42 &H74
&H26 &H20 &H22
&H31 &H64 &H0D

IAN SINCLAIR

Introducing Dragon Machine Code

Ian Sinclair

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data
Sinclair, Ian Robertson
Introducing Dragon machine code
1. Dragon (Computer)—Programming
I. Title
001.64'25 QA76.8.D

ISBN 0 246-12324-9

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by Mackays of Chatham, Kent

Contents

Preface

Many computer users are content to program in BASIC for all of their computing lives. A large number of others, however, are eager to find out more about computing and their computer than the use of BASIC can lead to. Few, however, make much progress to the use of machine code, which allows so much more control over the computer. The reason for this, to my mind, is that so many books which deal with machine code programming seem to start with the assumption that the reader is already familiar with the ideas and the words of such programming. In addition, many of these books treat machine code programming as a study in itself, leaving the reader with little clue as to how to apply machine code to his or her computer.

This book has two main aims. One is to introduce the Dragon owner to some of the details of how the Dragon works, so allowing for more effective programming even without delving into machine code. The second aim is to introduce the methods of machine code programming in a simple way. I must emphasise the word 'introduce'. No single book can tell you all about machine code, even the machine code for one computer. All I can claim is that I can get you started. Getting started means that you will be able to write short machine code routines, understand machine code routines that you see printed in magazines, and generally make more effective use of your Dragon. It also means that you will be able to make use of the many more advanced books on the subject.

Understanding the operating system of your Dragon, and having the ability to work in machine code can open up an entirely new world of computing to you. This is why you find that most of the really spectacular games are written in machine code. You will also find that many programs which are written mainly in BASIC will incorporate pieces of machine code in order to make use of its greater speed and better control of the computer.

I am most grateful to several people, all top-notch Dragon tamers, for discussions about details of the Dragon operating system and the 6809 microprocessor. These include Mr Opyrchal of Compusense, and Mike James, well known for his books and articles, and author of the *6809 Companion*. I must also thank some of the many people who made the production of this book possible. Of these, Richard Miles of Granada Publishing, and Robin Swinbank of Welbeck P. R. combined to get me a Dragon and keep it fed. Sue Moore and Julian Grover of Granada Publishing then did their usual miracles of meticulous effort on my manuscript. I should also mention that this book was completed before the Dragon assembler program, *DREAM*, was available, so that it was not possible to provide details of this product.

Ian Sinclair

Chapter One

ROM, RAM, Bytes and Bits

One of the things that discourage computer users from attempts to go beyond BASIC is the number of new words that spring up. The writers of many books on computing, especially on machine code computing, seem to assume that the reader has an electronics background and will understand all of the terms. I shall assume that you have no such background. All that I shall assume is that you possess a Dragon, either the 32 or the 64, and that you have some experience in programming your Dragon in BASIC. This means that we start at the correct place, which is the beginning. I don't want in this book to have to interrupt important explanations with technical or mathematical details, and these will be found in the Appendices. This way, you can read the full explanation of some points if you feel inclined, or skip them if you are not.

To start with, we have to think about *memory*. A unit of memory for a computer is, as far as we are concerned, just an electrical circuit that acts like a switch. You walk into a room, switch a light on, and you never think it's remarkable in any way that the light stays on until you switch it off. You don't go about telling your friends that the light circuit contains a memory – and yet each memory unit of a computer is just a kind of miniature switch that can be turned on or off. What makes it a memory is that it will stay the way it has been turned, on or off, until it is changed. One unit of computer memory like this is called a *bit* – the name is short for binary digit, meaning a unit that can be switched to one of two possible ways.

We'll stick with the idea of a switch, because it's very useful. Suppose that we wanted to signal with electrical circuits and switches. We could use a circuit like the one in Fig. 1.1. When the switch is on, the light is on, and we might take this as meaning YES. When the switch is turned off, the light goes out, and we might take

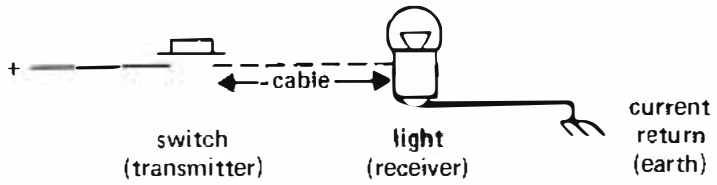
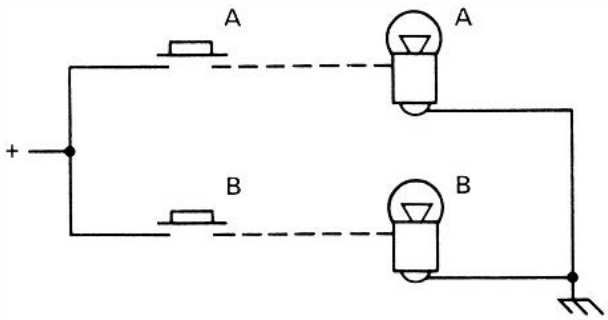


Fig. 1.1. A single-line switch and bulb signalling system



A	B
off	off
off	on
on	off
on	on

Fig. 1.2. Two-line signalling – four possible signals can be sent

this as meaning NO. You could attach any two meanings that you liked to these two conditions (called 'states') of the light, so long as there are only two. Things improve if you can use two switches and two lights, as in Fig. 1.2. Now four different combinations are possible: (a) both off, (b) A off, B on, (c) A on, B off, (d) both on. This set of four possibilities means that we could signal four different meanings. Using one line allows two possible codes; using two lines allows four codes. If you feel inclined to work them all out, you'll find that using three lines will allow eight different codes. A moment's thought suggests that since 4 is 2×2, and eight is 2×2×2, then four lines might allow 2×2×2×2, which is 16, codes. It's true, and since we usually write 2×2×2×2 as 2^4 (two to the power 4), we can find out how many codes could be transmitted by any number of lines. We would expect eight lines, for example, to be able to carry 2^8 codes, which is 256. A set of eight switches, then, could be arranged

so as to convey 256 different meanings. It's up to us to decide how we might want to use these signals.

One particularly useful way is called *binary code*. Binary code is a way of writing numbers using only two digits, Ø and 1. We can think of Ø as meaning 'switch off' and 1 as meaning 'switch on', so that 256 different numbers could be signalled by using eight switches by thinking of Ø as meaning off and 1 as meaning on. This group of eight is called a *byte*, and it's the quantity that we use to specify the memory size of our computers. This is why the numbers 8 and 256 occur so much in machine code computing.

The way that the individual bits in a byte are arranged so as to indicate a number follows the same way as we use to indicate a number normally. When you write a number such as 256, the 6 means six units, the 5 is written to the immediate left of the 6 and means five tens, and the 2 is written one more place to the left and means two hundreds. These positions indicate the importance or significance of a digit, as Fig. 1.3 shows. The 6 in 256 is called the 'least significant digit', and the 2 is the 'most significant digit'. Change the 6 to 7 or 5, and the change is just one part in 256. Change the 2 to 1 or 3 and the change is one hundred parts in 256 – much more important.

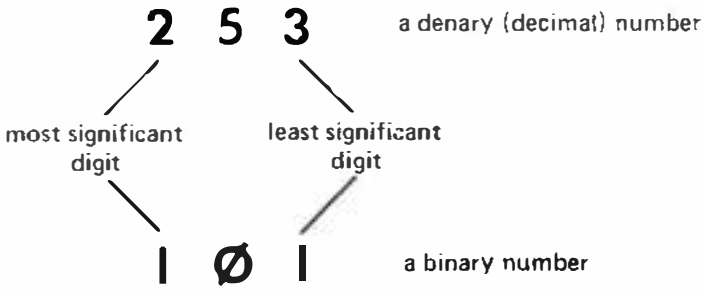


Fig. 1.3. The significance of digits. Our numbering system uses the position of a digit in a number to indicate its significance or importance.

Having looked at bits and bytes, it's time to go back to the idea of memory as a set of switches. As it happens, we need two types of memory in a computer. One type must be permanent, like mechanical switches or fixed connections, because it has to be used for retaining the number-coded instructions that operate the computer. This is the type of memory that is called ROM, meaning Read-Only Memory. This implies that you can find out and copy what is in the memory, but not delete it or change it. The ROM is the most important part of your computer, because it contains all the

instructions that make the computer carry out the actions of BASIC. When you write a program for yourself, the computer stores it in the form of another set of number-coded instructions in a part of memory that can be used over and over again. This is a different type of memory that can be 'written' as well as 'read', and if we were logical about it we would refer to it as RWM, meaning read-write memory. Unfortunately, we're not very logical, and we call it RAM (meaning Random-Access Memory). This was a name that was used in the very early days of computing to distinguish this type of memory from one which operated in a different way. We're stuck with the name of RAM now and probably forever!

The number-code caper

Now we can get back to the bytes. We saw earlier that a byte, which is a group of eight bits, can consist of any one of 256 different arrangements of these bits. The most useful arrangement, however, is one that we call *binary code*. These different arrangements of binary code represent numbers which we write in ordinary form as Ø to 255 (not 1 to 256, because we need a code for zero). Each byte of the 32768 bytes of RAM in the Dragon 32 can store a number which is in this range of Ø to 255.

Numbers by themselves are not of much use, and we wouldn't find a computer particularly useful if it could deal only with numbers between Ø and 255, so we make use of these numbers as codes. Each number code can, in fact, be used to mean several different things. If you have worked with ASCII codes in BASIC, you will know that each letter of the alphabet – and each of the digits Ø to 9 and each punctuation mark – is coded in ASCII as a number between 32 (the space) and 127 (the left-arrow). That selection leaves you with a large number of ASCII code numbers which can be used for other purposes such as graphics characters. The ASCII code is not the only one, however. Dragon uses its own coded meanings for numbers in this range of Ø to 255. For example, when you type the word PRINT in a program line, what is placed in the memory of the Dragon (when you press ENTER) is not the sequence of ASCII codes for PRINT. This would be 8Ø,82,73,78,84, one byte for each letter. What is put into memory, in fact, is one byte, the binary form of the number 135. This single byte is called a *token* and it can be used by the computer in two ways. One way is to locate the ASCII codes for the characters that make up the word PRINT. These are

stored in the ROM, so that when you LIST a program, you will see the word PRINT appear, not a character whose code is 135. The other, even more important, use of the token is to locate a set of instructions which are also held in the ROM in the form of number codes. These instructions will cause characters to be printed on the screen, and the numbers that make up these codes are what we call *machine code*. They control directly what the 'machine' does. That direct control is our reason for wanting to use machine code. When we use BASIC, the only commands we can use are the ones for which 'tokens' are provided. By using machine code, we can make up our own commands and do what we please. Incidentally, the fact that PRINT generates one 'token' is the reason why it is possible to use ? in place of PRINT. The Dragon has been designed so that a ? which is not placed between quotes will also cause 135 to be put into memory.

Do-it-yourself spot

As an aid to digesting all that information, try a short program. This one, in Fig. 1.4, is designed to reveal the keywords that are stored in

```
10 PRINT32819;" ";:FOR N=32819 TO 33089
20 K=PEEK(N)
30 IF K<128 THEN PRINTCHR$(K);
40 IF K>=128 THEN PRINTCHR$(K-128):PRINT
N;" ";
50 NEXT
```

Fig. 1.4. A program that reveals the keywords of Dragon BASIC.

the ROM, and it makes use of the BASIC instruction word PEEK. PEEK has to be followed by a number or number variable within brackets, and it means 'find what byte is stored at this address number'. All of the bytes of memory within your Dragon are numbered from zero upwards, one number for each byte. Because this is so much like the numbering of houses in a road, we refer to these numbers as *addresses*. The action of PEEK is to find what number, which must be between 0 and 255, is stored at each address. The Dragon automatically converts these numbers from the binary form in which they are stored into the ordinary decimal (more correctly, *denary*) numbers that we normally use. By using CHR$ in our program, we can print the character whose ASCII code is the number we have PEEKed at. The program uses the variable N as an

address number, and then checks that PEEK(N) gives a number less than 128 – in other words a number which is an ASCII code. If it is, then the character is printed.

Now the reason that we have to check is that the last character in each set of words, or word, is coded in a different way. The number that we find for the last character has had 128 added to the ASCII code. For example, the first three address locations that the program PEEKs at contain the numbers 7Ø, 79 and 21Ø. The number 7Ø is the ASCII code for F, 79 is the code for O, and then 21Ø–128=82, which is the ASCII code for R. This is where the word FOR is stored, then. The reason for treating the last letter so differently is to save memory! If a gap were left between words, this would be a byte of memory wasted. As it is, there is no waste, because the last letter of a group always has a code number that is greater than 128, so the computer can recognise it easily. We have followed the same scheme in the BASIC program of Fig. 1.4 by using line 4Ø to print the correct letter and to take a new line and print the address number. There is another set of numbers stored further on, which consists of more addresses. These are the addresses of subroutines which carry out the actions of BASIC, and they are stored in the same order as these words.

Dragon dissection

Now take a look at a diagram of the Dragon in Fig. 1.5. It's quite a simple diagram because I've omitted all of the detail, but it's enough

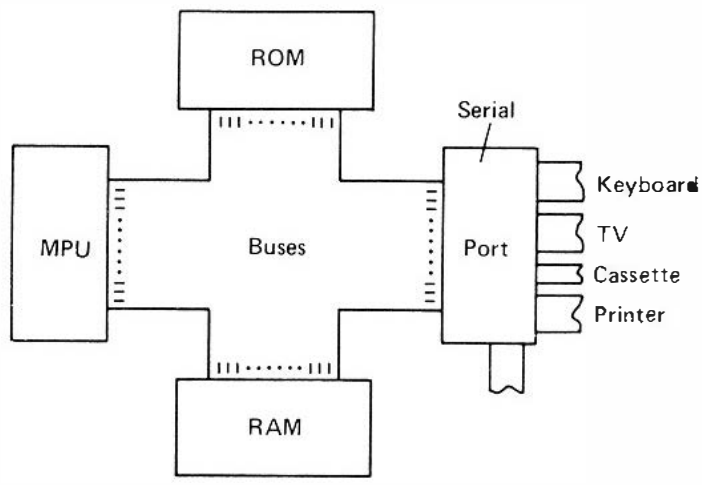


Fig. 1.5. A block diagram of Dragon. The connections marked *Buses* consist of a large number of connecting links which join all of the units of the system.

to give us a clue about what's going on inside. This is the type of diagram that we call a *block diagram*, because each unit is drawn as a block with no details about what may be inside. Block diagrams are like large-scale maps which show us the main routes between towns but don't show side-roads or town streets. A block diagram is enough to show us the main paths for electrical signals in the computer.

The names of two of the blocks should be familiar already, ROM and RAM, but the other two are not. The block that is marked MPU is a particularly important one. MPU means Microprocessor Unit – some block diagrams use the letters CPU (Central Processing Unit). The MPU is the main 'doing' unit in the system, and it is, in fact, one single unit. The MPU is a single plug-in chunk, one of those silicon chips that you read about, encased in a slab of black plastic and provided with 40 connecting pins that are arranged in two rows of 20 (Fig. 1.6). There are several different types of MPU made by different manufacturers, and the one in your Dragon is called 6809 (or 6809E). It's quite different from the MPU chips that are used in most other computers, so that books about other MPU chips such as

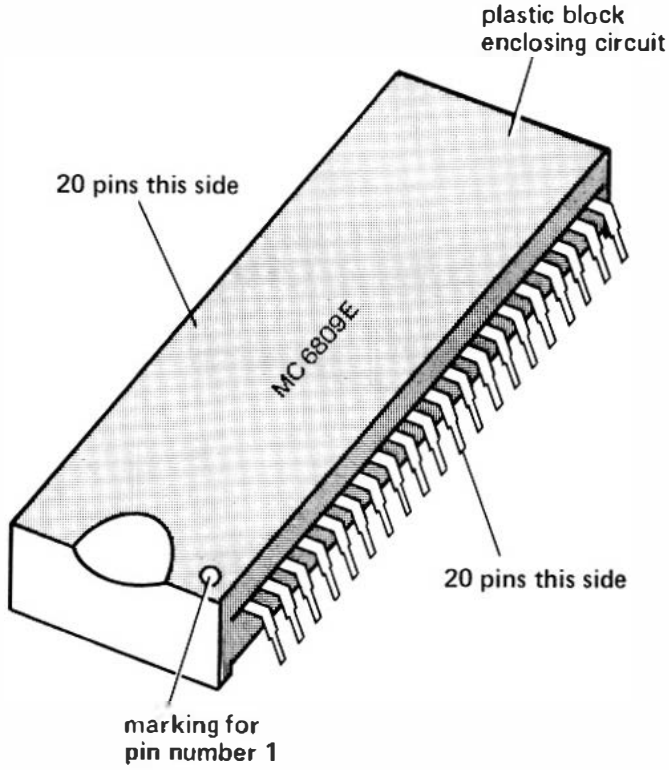


Fig. 1.6. The 6809 MPU. The actual working part is smaller than a fingernail, and the larger plastic case (52 mm long and 14 mm wide) makes it easier to work with.

the Z80 or the 6502 won't be of much help in understanding the 6809.

What does the MPU do? The answer is practically everything, and yet the actions that the MPU can carry out are remarkably few and simple. The MPU can load a byte, meaning that a byte which is stored in the memory can be copied into another store in the MPU. The MPU can also store a byte, meaning that a copy of a byte that is stored in the MPU can be placed in any address in the memory. These two actions (see Fig. 1.7) are the ones that the MPU spends

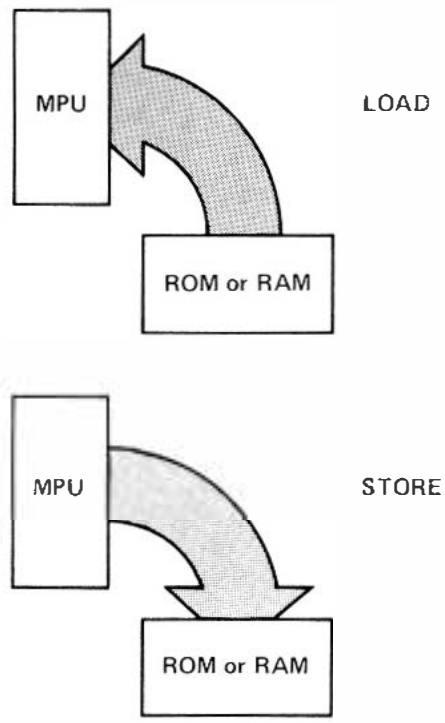


Fig. 1.7. Loading and storing. 'Loading' means signalling to the MPU from the memory, so that the digits of a byte are copied into the MPU. 'Storing' is the opposite process.

most of its working life in carrying out. By combining them, we can copy a byte from any address in memory to any other. You don't think that's very useful? That copying action is just what goes on when you press the letter H on the keyboard and see the H appear on the screen. The MPU treats the keyboard as one piece of memory and the screen as another, and copies bytes from one to the other as you type. That's a considerable simplification, but it will do for now just to show how important the action is.

Loading and storing are two very important actions of the MPU, but there are several others. One set of actions is the *arithmetic set*. For most types of MPU, these consist of addition and subtraction only. The 6809 is unique in its class in having a multiplication

instruction, but most of the arithmetic operations are simpler than this, and can use only single-byte numbers. Since a single-byte number means a number between Ø and 255, how does the computer manage to carry out actions like multiplication of large numbers, division, raising to powers, logarithms, sines, and all the rest? The answer is – by machine code programs that are stored in the ROM. If these programs were not there you would have to write your own. There aren't many computer users who would like to set about that task.

There's also the logic set. MPU logic is, like all MPU actions, simple and subject to rigorous rules. Logic actions compare the bits of two bytes and produce an 'answer' which depends on the bits' values that are compared and on the logic rule that is being followed. The three logic rules are called AND, OR and XOR, and Fig. 1.8 shows how they are applied.

Another set of actions is called the 'jump set'. A jump means a change of address, rather like the action of GOTO in BASIC. A combination of a test and a jump is the way that the MPU carries out its decision steps. Just as you can program in BASIC:

```
IF A = 36 THEN GOTO 1Ø5Ø
```

so the MPU can be made to carry out an instruction which is at an entirely different address from the normal next address. The MPU is a programmed device, meaning that it carries out each of its actions as a result of being fed with an instruction byte which has been stored in the memory. Normally when the MPU is fed with an instruction from an address somewhere (usually in ROM), it carries out the instruction and then reads the instruction byte that is stored in the next address up. A jump instruction would prevent this from happening. It would, instead, cause the MPU to read from another address, the one that was specified in the jump instruction. This jump action can be made to depend on the result of a test. The test will usually be carried out on the result of the previous action, whether it gave a zero, positive or negative result, for example.

That isn't a very long or exciting list, but the actions that I've omitted are either unimportant at this stage, or not particularly different from the ones in the list. What I want to emphasise is that the magical microprocessor isn't such a very smart device. What makes it so vital to the computer is that it can be programmed and that it can carry out its actions very quickly. Equally important is the fact that the microprocessor can be programmed by sending it electrical signals.

AND

The result of ANDing two bits will be 1 if both bits are 1, Ø otherwise:

```
1 AND 1 = 1   { 1 AND Ø = Ø }   Ø AND Ø = Ø
              { Ø AND 1 = Ø }
```

For two bytes, corresponding bits are ANDed

```
        1Ø11Ø111
AND     ØØØØ1111
        ØØØØØ111
```

only these bits exist in both bytes.

OR

The result of ORing two bits will be 1 if either or both bits is 1, Ø otherwise:

```
1 OR 1 = 1   { 1 OR Ø = 1 }   Ø OR Ø = Ø
             { Ø OR 1 = 1 }
```

For two bytes, corresponding bits are ORed

```
        1Ø11Ø111
OR      ØØØØ1111
        1Ø111111
```

only bit which is Ø in both.

XOR (Exclusive-OR)

Like OR, but result is zero if the bits are identical

```
1 XOR 1 = Ø   { 1 XOR Ø = 1 }   Ø XOR Ø = Ø
              { Ø XOR 1 = 1 }
```

```
        1Ø11Ø111
XOR     ØØØØ1111
        1Ø111ØØØ
```

if two bits are identical the result is zero.

Fig. 1.8. The rules for the three logic actions, AND, OR and XOR.

These signals are sent to eight pins, called the *data pins*, of the MPU. It doesn't take much of a guess to realise that these eight pins correspond to the eight bits of a byte. Each byte of the memory can therefore affect the MPU by sharing its electrical signals with the MPU. Since this is a long-winded description of the process, we call it 'reading'. 'Reading' means that a byte of memory is connected along eight lines to the MPU, so that each 1 bit will cause a 1 signal on a data pin, and each 0 bit will cause a 0 signal on a data pin. Just as reading a paper or listening to a recording does not destroy what is written or recorded, reading a memory does not change the memory in any way, and nothing is taken out. The opposite process of writing does, however, change the memory. Like recording a tape, writing wipes out whatever existed there before. When the MPU writes a byte to an address in the memory, whatever was formerly stored at that address is no longer there – it has been replaced by the new byte. This is why it is so easy to write new BASIC lines replacing old ones at the same line number.

Table d'Hôte?

Do you really write programs in BASIC? It might sound like a silly question, but it's a serious one. The actual work of a program is done by coded instructions to the MPU, and if you write only in BASIC, you don't write these. All that you do is to select from a menu of choices that we call the BASIC keywords, and arrange them in the order that you hope will produce the correct results. Your choice is limited to the keywords that are designed into the ROM. We can't alter the ROM. Therefore, if we want to carry out an action that is not provided for by a keyword, we must either combine a number of keywords (a BASIC program) or operate directly on the MPU with number codes (machine code). When you have to carry out actions by combining a number of BASIC commands, the result is clumsy, especially if each command is a collection of other commands. Direct action is quick, but it can be difficult. The direct action that I am talking about is machine code, and a lot of this book will be devoted to understanding this 'language' which is difficult just because it's simple!

Take a situation which will illustrate this paradox. Suppose you want a wall built. You could ask a builder. Just tell him that you want a wall built across the back garden, and then sit back and wait. This is like using BASIC with a command word for 'build a wall'.

There's a lot of work to be done, but you don't have to bother about the details.

Now think of another possibility. Suppose you had a robot which could carry out instructions mindlessly but incredibly quickly. You couldn't tell it to 'build a wall' because these instructions are beyond its understanding. You have to tell it in detail, such as: 'stretch a line from a point 85 feet from the kitchen edge of the house, measured along the fence southwards, to a point 87 feet from the lounge end of the house measured along that fence southwards. Dig a trench eighteen inches deep and one foot wide along the path of your line. Mix three bags of sand and two of cement with four barrow-loads of pebbles for three minutes. Mix water into this until a pail filled with the mixture will take ten seconds to empty when held upside down. Fill the trench with the mixture ...'. The instructions are very detailed – they have to be for a brainless robot – but they will be carried out flawlessly and quickly. If you've forgotten anything, no matter how obvious, it won't be done. Forget to specify how much mortar, what mixture and where to place it, and your bricks will be put up without mortar. Forget to specify the height of the wall, and the robot will keep piling one layer on top of another, like the Sorcerer's Apprentice, until someone sneezes and the whole wall falls down.

The parallel with programming is remarkably close. One keyword in BASIC is like the 'build a wall' instruction to the builder. It will cause a lot of work to be done, drawing on a lot of instructions that are not yours – but it may not be done as fast as you like. If you can be bothered with specifying the detail, machine code is a lot faster because you are giving your instructions direct to an incredibly fast but mindless machine, the microprocessor. We can stretch the similarity further. If you said to your builder 'mend the car', he might be unwilling or unable to do so. The correct set of detailed instructions to the robot would, however, get this job done. Machine code can be used to make your computer carry out actions that are simply not provided for in BASIC, though it's fair to say that many modern computers allow a much greater range of commands than early models, and this aspect of machine code is not quite so important as it used to be.

One last look at the block diagram is needed before we start on the inner workings of the Dragon. The block which is marked 'Port' includes more than one chip. A port in computing language means something that is used to pass information, one byte at a time, into or out from the rest of the system – the MPU, ROM and RAM. The

reason for having a separate section to handle this is that inputs and outputs are important but slow actions. By using a port we can let the microprocessor choose when it wants to read an input or write an output. In addition, we can isolate inputs and outputs from the normal action of the MPU. This is why nothing appears on the screen in a BASIC program except where we have a PRINT command in the program. It's also why pressing the PLAY key of the cassette recorder has no effect until you type CLOAD and press ENTER. The port keeps the action of the computer hidden from you until you actually need to have an input or an output.

We have now looked at all of the important sections of your Dragon. I've used some terms loosely – purists will object to the way I've used the word 'port', for example – but no-one can quarrel with the actions that are carried out. What we have to do now is to look at how the computer is organised to make use of the MPU, ROM, RAM and ports so that it can be programmed in BASIC and can run a BASIC program. It looks like a good place to start another chapter!

Chapter Two

Digging Inside Dragon

I don't mean 'digging inside' literally – you don't have to open up the case. What I do mean is that we are going to look at how the Dragon is designed to load and run BASIC programs. We'll start with a simplified version of the action of the whole system, omitting details for the moment.

The ROM of your Dragon, which starts at address 32768, consists of a large number of short programs – *subroutines* – which are written in machine code, along with sets of values (tables) like the table of keywords. There will be at least one machine code subroutine for each keyword in BASIC, and some of the keywords may require the use of many subroutines. When you switch on your Dragon, the piece of machine code that is carried out is called the *initialisation routine*. This is a long piece of program but, because machine code is fast, carrying out instructions at the rate of many thousands per second, you see very little evidence of all this activity. All that you notice is a delay between switching on and seeing the MICROSOFT copyright notice. In this brief time, though, the action of the RAM part of the memory has been checked, some of the RAM has been 'written' with bytes that will be used later, and most of the RAM has been cleared for use. 'Cleared for use' does not mean that nothing is stored in the RAM. When you switch off the computer, the RAM loses all trace of stored signals, but when you switch on again the memory cells don't remain storing zeroes. In each byte, some of the bits will switch to 1 and some will switch to Ø when power is applied. This happens quite at random, so that if you could examine what was stored in each byte just after switching on, you would find a set of meaningless numbers. These would consist of numbers in the range Ø to 255, the normal range of numbers for a byte of memory. These numbers are 'garbage' – they weren't put into memory deliberately, nor do they form useful instructions or data. The first job of the computer, then, is to clean up. In place of the

random numbers, the computer substitutes a very much more ordered pattern of 4 bytes of 255 followed by four bytes of Ø. Try this – switch on, and type (no line number):

```
FOR N =13824 TO 13847:?PEEK(N);" ";:NEXT
```

and then press ENTER. The range of memory addresses we have used is the 'start of BASIC' range, where the first bytes of a BASIC program are normally stored. If we have just switched on, and haven't used a line number for the command, there will be nothing stored here except for the pattern that was left after the initialisation. As you'll see on the screen, it consists of the chain of 255's and Ø's.

The initialising program has a lot more to do. The first section of RAM, from address Ø to 1Ø23, is for 'system use'. This is because the machine code subroutines which carry out the actions of BASIC need to store quantities in memory as they are working. Address numbers 25 and 26, for example, hold the address of the first byte of a BASIC program. A much larger section of the memory is used for storing numbers that make text and graphics appear on the screen. In addition, some RAM has also to be used to hold quantities that are created when a program runs. That's what we are going to look at now.

Variables on the table

BASIC programs make a lot of use of variables, meaning the use of letters to represent numbers and words. Each time you 'declare a variable' by using a line like:

```
N=2Ø   or   A$="SMITH"
```

the computer has to take up memory space with the name (N or A$ or whatever you have used) and the value (like 2Ø or SMITH) that you have assigned to it. The piece of memory that is used to keep track of variables is called the variable list table (VLT). It doesn't occupy any fixed place in the memory, but is stored in the free space just above your program. If you add one more line to your program, the VLT address has to be moved to a set of higher address numbers. If you delete a line from your program, the VLT will be moved down in the same way so that it is always kept just following the last line of BASIC.

Now because the variable list table address can and does move around as the program is altered, the computer must at all times

keep a note of where the table starts. This is done by using one of the pieces of memory that are reserved for system use, the addresses 27 and 28. You may wonder why two addresses are used. The reason is that one byte can hold a number only up to 255 in value. If we use two bytes, however, we can hold the number of 256's in one byte and the remainder in the other. A number like 257, for example, is one 256 and one remaining. We could code this as 1,1. This means that a 1 is stored in the byte that is reserved for 256's, and 1 in the byte reserved for units. The order of storing the numbers is high-byte then low-byte. To find the number that is stored, we multiply one byte by 256 and add the other. For example, if you found 3,56 stored in two consecutive addresses that were used in this way, this would mean the number:

3*256 + 56 = 824

The biggest number that we can store using two bytes like this is 255,255, which means 255*256+255=65535. This is the reason that you can't use very large numbers like 7ØØØØ as line numbers in the Dragon – the operating system uses only two bytes to store its line numbers. In fact, for other reasons, the maximum number that you can use is 63999.

All of this means that we can find the address that is stored in addresses 27 and 28 by using the formula:

?PEEK(27)*256+PEEK(28)

If you use this just after you have switched on your Dragon, then the result on the 32K Dragon is the address number 7683. This is just above the address at which the first byte of a BASIC program would be stored. To see this in action, type the line:

1Ø N=2Ø

and try ?PEEK(27)*256+PEEK(28) again. If you typed this line as I did, with a space between the line number and the 'N', then the address that you get is 7692. The variable list table has moved upwards in the memory, by 9 bytes. That's more than the number of bytes that you typed, you'll notice – reasons later.

Quite a lot of important addresses that the computer uses are 'dynamically allocated' like this. 'Dynamically allocated' means that the computer will change the place where groups of bytes are to be kept. It will then keep track of where they have been stored by altering an address that is held in a pair of bytes such as this example. This has important implications for how you use your computer.

For example, if you shift the VLT by poking new numbers into addresses 27 and 28, the computer can't find its variable values. Try this – after finding the VLT address, but without running the one-line program, 1Ø N=2Ø , type ?N . The answer will be zero. Why? Because the program has not been run. The address 7692 is where the VLT will start, but there's no VLT created until the program runs. This makes it easy for you to add or delete lines at this stage. All that will have to be altered is the pair of numbers in addresses 27 and 28. The VLT values are put in place only when the program runs, and the table is never actually moved. All that changes when you edit is the starting address in 27 and 28. That's why you can't resume a program after editing – you have to RUN again to create a new VLT at a new address. If you RUN the one-line program now, and then type ?N you will get the expected answer of 2Ø. Now type (no line number) POKE 27,5Ø, and press ENTER. Try ?N and see what you get. On my Dragon, it was Ø again, because the correct value of variable N can be found no longer. If your Dragon locks up during this exercise, then the SYSTEM RESET button will restore control, but the program may be lost. Note, incidentally, the use of POKE to place a new value into a memory address. The correct form of the command is POKE A,D . A is an address, and will be in the range Ø to 32767 (the range of values of RAM memory) for the 32K Dragon. D is the data that you place into this memory address, and it must be a value between Ø and 255. If you try to poke a number greater than 255, you will get an 'FC ERROR' message instead.

A look at the table

It's time now to do something more constructive, and take a look at what is stored in the VLT. When we do these investigations, it's important to ensure that the computer is clear of the results of previous work, so it's advisable to switch off and then on again, before each effort. Simply pressing the SYSTEM RESET button does not alter values that you may have poked into the memory. It's tedious, I know, but that's machine code for you!

To work, then. After switching off and on again, type the line:

```
1Ø N=2Ø
```

again, and find the VLT address by using:

```
?PEEK(27)*256+PEEK(28)
```

```
7692    78
7693    0
7694    133
7695    32
7696    0
7697    0
7698    0
7699    88
7700    0
7701    141
7702    112
```

Fig. 2.1. The variable list table entry for a simple number variable.

This gave me the address 7692 again. Now type RUN, so that values are put into the VLT, and take a look at what has been stored there. This is done by using the command:

```
FORX=7692 TO 77Ø2:?X;"  ";PEEK(X):NEXT
```

and pressing ENTER. This gives the listing that is illustrated in Fig. 2.1. Now can we recognise anything here? We ought to recognise the first byte of 78, because that's the ASCII code for N! The next byte is zero because our variable is called N, not NI or NG or any other two-letter name. If we used a two-letter name, then both addresses 7692 and 7693 would have been occupied. The next five bytes, then, must be the way that the number 2Ø has been coded. At this point, don't worry about how these numbers are used to represent 2Ø – just accept that they do! How do I know that it's the next five bytes that represent the number 2Ø? Easy, the byte in address 7699 is 88, which is the ASCII code for X, and that's the variable we used to print the table values! The Dragon always uses just five bytes for any value of number variable, no matter whether it's a small number like 2Ø, or a very much larger one like 1427Ø68315, or a fraction, or negative. This makes the storage of number variables simple, and it also makes it easy for the computer to find variables. If, for example, it is looking for the value of a variable called Y, then when it finds 'N' (coded as ASCII 78) it need not waste time with the next six bytes (one for a second letter, five for the value), but moves to the next place where a variable name will be stored. If you are curious, and have a head for mathematics, Appendix A shows what method of coding is used to convert numbers into five bytes. For the purposes of this book you don't, however, need to understand how the coding is done as long as you know how the code is stored and how many bytes are needed.

Tying up the string

Now we need to take a look at how a string variable is stored. Switch off and on again, and then type the line:

```
10 AB$="THIS IS A STRING"
```

RUN this one-liner, and then find the VLT address by using addresses 27 and 28 as before. I obtained 7710 for this. Now use:

```
FOR X=7710 TO 7720:?X;"  ";PEEK(X):NEXT
```

to find what is in the VLT. This time, it's as Fig. 2.2 shows. The first

```
7710   65
7711   194
7712   16
7713   0
7714   30
7715   10
7716   0
7717   88
7718   0
7719   141
7720   113
```

Fig. 2.2. The VLT entry for a string variable.

value in this table is 65, which is the ASCII code for A. The second, however, is 194. Now this is the ASCII code for B with 128 added to it, and it's the way that the Dragon recognises that this is a string variable. If you had used the variable name A$ rather than AB$, then the second number (at address 7711) would have been 128, not 0. When you use a number variable, the second ASCII code of the name will be 0, or one of the ASCII code numbers, never greater than 127. Good thinking, designers!

Now take a look at the rest of the entry for this string. It doesn't look much like the ASCII codes for the letters, does it? In fact, the entry consists of seven bytes only, just the same total length as a number variable. The clue to what is being done emerges when we take a look at the numbers. The number that follows the code for B (194, because 128 has been added to the ASCII code) is 16. Now 16 is the number of characters in the string. If you count the number of letters and spaces you'll see that this is what it comes to. The next byte is zero, and then there are two bytes, 30 and 10. Now two bytes together are always likely to be an address, and if we combine them in the usual way, using 30*256+10, we get 7690. Next step in the

trail is PEEK(769Ø). Sure enough, it's 84, the ASCII code for 'T'. 769Ø is the address of the first byte of the string.

Let's gather all this up. The Dragon stores an entry of seven bytes in its VLT for each string. Of these seven bytes, the first two are for the string name, and the second will be 128 or more. When a two character name is used, 128 is added to the ASCII code for the second letter. This allows the computer to distinguish a string variable from a number variable. The next five bytes then contain the length of the string and the address in memory of its first byte. As it happens, only three bytes are needed to keep track of a string. One byte is needed for the length – no string will exceed 255 characters(in fact you are not allowed to enter more than 241). Two bytes are needed for the address, so that two of the seven bytes that are used in the string VLT entry are not used except as separators. The convenience of having the same total length of VLT entry for a string as for a number outweighs the slight waste of two bytes in each string entry.

In this example, the string is stored at an address lower than the VLT, in the 'BASIC text' part of the memory. This is the part of the memory which contains the program, and since the ASCII codes for the string are placed here when you type the program, it's as good a resting place as any. Numbers must be transferred to the VLT

7712	65
7713	128
7714	2
7715	Ø
7716	3Ø
7717	9
7718	Ø
7719	66
7720	128
7721	2
7722	Ø
7723	3Ø
7724	17
7725	Ø
7726	67
7727	128
7728	4
7729	Ø
7730	127
7731	184
7732	Ø
7733	88

Fig. 2.3. The VLT entry for a string which is not stored in the program part of memory.

because they are not stored in ASCII codes. The question now is, what happens when a string is created which does not exist in the program? If you type, for example:

```
10 A$="AB":B$="CD":C$=A$+B$
```

and RUN this, you will find that your VLT is longer, as you might expect. You will have to look at memory addresses from 7712 to 7733 this time. You will find the entries for A$ and B$, just as you would expect, giving addresses inside the program memory region, as shown in Fig. 2.3. The variable C$, however, gives the bytes 127,251 for its address. This corresponds to an address of 32763 (it's 127*256 + 251, remember) for this string. We can take a look at these addresses. If you type:

```
FORX=32763TO32766:?X"  ";PEEK(X);"  ";CHR$(PEEK
(X)):NEXT
```

then all will be revealed. The ASCII codes for letters ABCD are now stored here, and the use of CHR$ in the program reveals them.

Pointing the way

As it happens, your Dragon has a BASIC command which allows addresses to be obtained from the VLT. The command is VARPTR, and it has to be followed by the name of the variable within brackets. If you now type: ?VARPTR(A$) and press ENTER, you will find 7714 on the screen. This is NOT the address of the variable, it's the address of its VLT entry. VARPTR gives the address of where the length of the string is stored, ignoring the name bytes. To find the length byte, you need to use ?PEEK(VARPTR(A$)). If you want the address, you have to go to the second and third bytes following the length byte. For example, you can use:

```
?PEEK(VARPTR(A$)+2)*256+PEEK(VARPTR(A$)+3)
```

which will give 7689 as the string address as before. This allows some smart dodges inside BASIC programs, like swapping the names of string variables!

Program time

It's time now to look at how a program is stored in the memory of

```
10 A=10
20 PRINT A
30 C$="DRAGON"
```

Fig. 2.4. A simple BASIC program

your Dragon. As before, we shall rely on PEEKs at parts of the memory to find out what is happening. The first thing we need to know, however, is where the bytes that form the address of the start of a program are stored. As it happens, they are stored at 25 and 26.

We can therefore start looking at a program as it exists in the memory. Type the program as shown in Fig. 2.4, but don't run it.

```
7681    30
7682    10
7683    0
7684    10
7685    65
7686    203
7687    49
7688    48
7689    0
7690    30
7691    18
7692    0
7693    20
7694    135
7695    32
7696    65
7697    0
7698    30
7699    34
7700    0
7701    30
7702    67
7703    36
7704    203
7705    34
7706    68
7707    82
7708    65
7709    71
7710    79
7711    78
7712    34
7713    0
7714    0
7715    0
7716    78
```

Fig. 2.5. The bytes that represent the program in memory

Now type:

```
?PEEK(25)*256+PEEK(26)
```

and you will find the address at which the first byte of this program starts. In this example, my Dragon gave the address 7681. Now when you use the usual loop to print values of the PEEK numbers from this address onwards, you get the list as shown in Fig. 2.5. At first sight it looks like a stream of meaningless numbers but, when you look more carefully, you can see some pattern in it. As usual, the ASCII codes act as useful signposts. At 7685, for example, you can see the number 65, which is the ASCII code for 'A'. Since we know that the line is 'A=1Ø', we can look for the rest of this line. The 1Ø is recognisable as 49 (ASCII '1') and 48 (ASCII 'Ø'), so that the number 2Ø3 must represent the '=' sign. Now this is *not* the ASCII code for '=', but one of those 'tokens' that I mentioned in Chapter 1. It's a token because the computer is required to carry out an action, not just store an ASCII code here. The Ø at address 7689 marks the end of this line.

Now we have to grapple with the first four bytes. The first two are, as you might suspect from looking at them, an address. The 3Ø, 1Ø makes up the address 769Ø. What is this address? Why, it's the address of the first byte of the next line! This is how the operating system of the Dragon can pick out lines, and put them into the correct sequence, no matter what order you use to enter them. The final mystery is easily solved. Looking at the third and fourth bytes of each of the lines shows the sequence 1Ø, 2Ø, 3Ø – the line numbers. There are two bytes reserved for the line numbers because we want to have line numbers higher than 255. For line numbers smaller than 256, the first of these bytes, the more significant byte, is not used.

Now take a look at the other lines, as they appear stored in the memory. We have met the PRINT token of 135 before, and all the rest should be familiar by now. The only novelty is the end of the program. The last line ends with a Ø as usual, but following it, in the place where the address bytes for the next line would be, is another pair of zeros. This is the marker that the computer uses for END.

We can carry out some interesting changes on a program like this. Suppose, for example, that we poke the addresses that are used to carry the line numbers. If you type:

```
POKE7693,1Ø:POKE77Ø3,1Ø
```

and press ENTER, you will have placed the number 1Ø in each line number address for the lines 2Ø and 3Ø. Now LIST and look at the

result! It's a program of line 1Ø's. Contrary to what you might expect, this will RUN perfectly normally. The action of running, you see, depends on the 'next line' addresses being correct, not on how the lines are numbered. A program that has been altered in this way, however, is certainly not normal. If you try to edit, for example, there is only one line number to edit, and you'll get only the first line when you try EDIT 1Ø. That's because the computer starts looking at line numbers from the start of the program and needs to look no further than the first line. You can, however, record a program which has been altered in this way, and replay it normally, and RENUM will operate normally to restore the original line numbers.

Running the program

Now that we have looked at the way in which a program is coded and stored in the memory of the Dragon, we can give a bit of thought as to how it runs. This action is carried out by the most complicated part of the operating system, and it has to be given a starting address. This address comes, as you might expect, from the locations 25 and 26 which we have used. Suppose we go through the actions, omitting detail, of the three line program of Fig. 2.4. At the first address in BASIC, the RUN subroutine will read the first two bytes, and store them temporarily. These bytes will be used in place of the 'start of BASIC' address when the next line is carried out. The line number bytes are then read and stored. Why? So that if there is a syntax error in the line, the computer will be able to print out the message: 'SN ERROR IN 1Ø' rather than 'SN ERROR SOMEWHERE'! The next byte is an ASCII code, and the computer will take this as being a variable name. In the old days, the word LET had to be used to 'declare a variable'. This required another token, and most modern designs have dispensed with LET on the grounds that it's just as easy to arrange the subroutines to insert a LET token for a letter that immediately follows a line number. This means that, if you put a number in this place, it will be regarded as being part of the line number. The special token for the '=' sign then causes a subroutine to swing into action. This one creates an entry in the variable list table, at the first available address, and puts the ASCII code for A in that place. The next address in the VLT is left blank – there's no second letter for this variable name. The number 1Ø is then read and converted to the special binary form, as noted in Appendix A. This set of bytes is also placed in the VLT as the entry for A. The next byte

of the program is then read. It's Ø, so that the address for the next line, which was read as the first action, is now placed into the microprocessor. The type of action that we have considered in detail for line 1Ø then repeats with line 2Ø. This time, more has to be done when the action token is read. Since this is the token for PRINT, the subroutine for PRINT must be called up. It will locate the address of the next vacant place on the screen. This is done by keeping a note of the address in a couple of bytes of RAM – read these bytes, and you have the address. The value of A is then found in the VLT, and the bytes converted back to ASCII code form. The codes are then placed, one by one, in the screen memory. Doing this causes the characters to become visible on the screen, because of another subroutine. Once again, the zero at the end of the line causes the next line number to be used. At the end of the third line, however, the 'next line' number is zero, and the program ends. The computer goes back to its waiting state, ready for another command.

It's not quite so simple as that description makes it sound, but the essentials are there. The important thing to realise is that there is a lot of action to be done, and it has to be done one step at a time. What makes BASIC slow is that each token calls up a subroutine, which has to be found. For example, if you have a program that consists of a loop like:

```
1Ø FOR N=1 TO 5Ø
2Ø PRINT N
3Ø NEXT
```

then the action of reading the PRINT token of 135, and finding where the correct subroutine is stored, will be carried out 5Ø times. There is no simple way of ensuring that the subroutine is located once and then just used 5Ø times. The kind of BASIC that you have on your Dragon is 'interpreted' BASIC, which means that each instruction is worked out as the computer comes to it. If that means finding the address of the PRINT subroutine 5Ø times, so be it. The alternative is a scheme called *compiling*, in which the whole program is converted to efficient machine code before it is run. Compiling is done using another program, called a *compiler*. At present, I don't know of any compiler for the Dragon, but there may be one available by the time you read this!

Chapter Three

The Microprocessor

In this chapter, we'll start to get to grips with the 6809 microprocessor of the Dragon. The microprocessor, or MPU is, you remember, the 'doing' part of the computer as distinct from the storing part (memory) or the input/output part (ports). Consequently, what the microprocessor does will control what the rest of the computer does.

The MPU itself consists of a set of memory stores for numbers, but with a lot of organisation added. By means of circuits that are very aptly called *gates*, the way in which bytes are transferred between different parts of the MPU's own memory can be controlled, and it is these actions that constitute the addition, subtraction, logic and other actions of the MPU. Each of the actions is programmed. Nothing will happen unless an instruction byte is present in the form of a 1 or a Ø signal at each of the eight data terminals, and these bytes are used to control the gates inside the MPU. What makes the whole system so useful is that because the program instructions are in the form of electrical signals on eight lines, these signals can be changed very rapidly. The speed is decided by another electrical circuit called a *clock-pulse generator*, or *clock* for short. The speed that has been chosen as standard for the clock of the Dragon is rather slow, but it can be changed! The change is carried out by poking Ø into address 65495. When you do this, you can type faster without missing letters, your loops run faster, and your animated characters move faster. There is one drawback – you cannot use the cassette CLOAD and CSAVE commands reliably. To revert to normal speed for these commands, POKE 65494,Ø and press ENTER. Normal speed is less than a million clock pulses per second. It may not seem slow to you, but at a time when many computers use clock speeds of four million pulses per second or faster, the Dragon clock runs quite slowly!

Machine code

The program for the MPU, as we have seen, consists of number codes, each being a number between Ø and 255 (a single byte number). Some of these numbers may be instruction bytes which cause the MPU to do something. Others may be data bytes, which are numbers to add, or store or shift, or which may be ASCII codes for letters. The MPU can't tell which is which – it simply does as it is instructed. It's up to the programmer to sort out the numbers and put them into the correct order.

The correct order, as far as the MPU is concerned, is quite simple. The first byte that is fed to the MPU after switching on or completing an instruction, is taken as being an instruction byte. Now many of the 6809 instructions consist of just one byte, and need no data. Others may be followed by one or two bytes of data, and some instructions need two bytes. When the MPU reads an instruction byte, then it analyses the instruction to find if the instruction is one that has to be followed by one or more other bytes. If, for example, the instruction byte is one that has to be followed by two data bytes, then when the MPU analyses the first byte, it will treat the next two bytes that are fed to it as being the data bytes for that instruction. This action of the MPU is completely automatic, and is built into the MPU. The snag is that the machine code programmer must work to the same rules and, to get the program right, 100% correct is just about good enough. If you feed a microprocessor with an instruction byte when it expects a data byte or with a data byte when it expects an instruction byte, then you'll have trouble. Trouble nearly always means an endless loop, which causes the screen to go blank and the keys to have no effect. Even the SYSTEM RESET button can sometimes fail to break the Dragon out of such a loop, and the only remedy is to switch off. You will generally lose whatever program you had in store, so that it's vitally important to save any machine code program or a BASIC program that causes machine code actions (by using POKE) on tape before you use it.

What I want to stress at this point is that machine code programming is tedious. It isn't necessarily difficult – you are drawing up a set of simple instructions for a simple machine – but it's often difficult for you to remember how much detail is needed. When you program in BASIC, the machine's error messages will keep you right, and help to detect mistakes. When you use machine code you're on your own, and you have to sort out your own mistakes. In this respect, a program called an *assembler* helps

considerably. We'll look at that point again later. In the meantime, since the best way to learn about machine code is to write it, use it, and make your own mistakes. We'll start looking at how this is done, and we'll begin with the ways of writing the numbers that constitute the bytes of a machine code program.

Binary, denary and hex

A machine code program consists of a set of number codes. Since each number code is a way of representing the 1's and Ø's in a byte, it will consist of numbers between Ø and 255 when we write it in our normal scale of ten (denary scale). The program is useless until it is fed into the memory of the Dragon, because the MPU is a fast device, and the only way of feeding it with bytes as fast as it can use them is by storing the bytes in the memory, and letting the MPU help itself to them in order. You can't possibly type numbers fast enough to satisfy the MPU, and even methods like tape or disk are just not fast enough.

Getting bytes into the memory, then, is an essential part of making a machine code program work, and we shall look at methods in more detail later on. At one time, simple and very short programs would be put into a memory by the most primitive possible method, using eight switches. Each switch could be set to give a 1 or Ø electrical output, and a button could be pressed to cause the memory to store the number that the switches represented, and then select the next memory address. Programming like this is just too tedious, though, and working with binary numbers of 1's and Ø's soon makes you cross-eyed. Now that we have computers, it makes sense to use the computer itself to put numbers into memory, and an equally obvious step is to use a more convenient number scale.

Just what is a convenient number scale is a matter that depends on how you enter the numbers and how much machine code programming you do. The Dragon contains subroutines which convert the binary numbers in its memory to the form of denary numbers to print on the screen, and will also carry out the reverse action. When you use PEEK, the address that you want can be written in denary, and the result of the PEEK will be a number in denary, between Ø and 255. When you use POKE, similarly, you can type both the address number and the byte to be poked in denary.

Serious machine code programmers, however, find the use of denary anything but convenient. A denary number for a byte may be

one figure (like 4) or two (like 17) or three (like 143). A much more convenient code is the one called *hex* (short for hexadecimal) code. All one-byte numbers can be represented by just two hex digits. In conjunction with this, serious machine code programmers write their programs in what is called assembly language. This uses command words which are shortened versions of the names of commands to the MPU. Programs that are called assemblers then convert these command words into the correct binary codes. Practically all assemblers show these codes on the screen in hex form rather than in denary. In addition, when you type data numbers, you will have to make use of hex code. Since the Dragon also contains routines for working in hex, it seems sensible to learn how to use this facility. Not only does it make it easier for you to progress in machine code, it allows you to make effective use of some excellent software for developing machine code, such as DASM and DEMON (see Appendix B). 'Hexadecimal' means scale of sixteen, and the reason that it is used so extensively is that it is naturally suited to representing binary bytes. Four bits, half of a byte, will represent numbers which lie in the range Ø to 15 in our ordinary number scale. This is the range of one hex digit (Fig. 3.1.). Since we don't have symbols for digits higher than 9, we have to use the letters, A,B,C,D,E, and F to supplement the digits Ø to 9 in the hex scale. The advantage is that a byte can be represented by a two-digit number, and a complete address by a four-digit number. The number codes that are used as instructions have been designed in hex code, so that we can see much better how commands are related. For example, we may find that a set of related commands all start

Hex	Denary	Hex	Denary
Ø	Ø	C	12
1	1	D	13
2	2	E	14
3	3	F	15
4	4	then	
5	5	1Ø	16
6	6	11	17
7	7	to	
8	8	2Ø	32
9	9	21	33
A	1Ø	22	34
B	11	etc.	

Fig. 3.1. Hex and denary digits

with the same digit when they are written in hex. In denary, this relationship would not appear. In addition, it's much easier to write down the binary number which the computer actually uses when you see the hex version. The use of the Dragon assembler and monitor programs, such as the excellent DASM and DEMON, demand familiarity with hex, and books of information on the 6809 MPU will all have been written assuming that you know hex. It sounds as if we ought to make a start on it!

The hex scale

The hexadecimal scale consists of sixteen digits, starting as usual with Ø and going up in the usual way to 9. The next figure is not 1Ø, however, because this would mean one sixteen and no units, and since we aren't provided with symbols for digits beyond 9, we use the letters A to F. The number that we write as 1Ø (ten) in denary is written as ØA in hex, eleven as ØB, twelve as ØC and so on up to fifteen, which is ØF. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four even if fewer digits are needed. The number that follows ØF is 1Ø, sixteen in denary, and the scale then repeats to 1F, thirty-one, which is followed by 2Ø. The maximum size of byte, 255 in denary, is FF in hex. When we write hex numbers, it's usual to mark them in some way so that you don't confuse them with denary numbers. There's not much chance of confusing a number like 3E with a denary number, but a number like 26 might be hex or denary. The convention that is followed by 6809 programmers is to use the dollar sign ($) to mark a hex number, with the sign placed before the number. For example, the number $47 means hex 47, but plain 47 would mean denary forty-seven. When you write hex numbers for a 6809 program, it's advisable to follow this convention. The Dragon, however, uses a different method when you want to poke hex numbers. The Dragon method is &H, so that POKE&H19,&HF would poke the hex address of 19 with the hex number F.

Now the great value of hex code is how closely it corresponds to binary code. If you look at the hex-binary table of Fig. 3.2, you can see that $9 is 1ØØ1 in binary and $F is 1111. The hex number $9F is therefore just 1ØØ11111 in binary – you simply write down the binary digits that correspond to the hex digits. The conversion in the opposite direction is just as easy – group the binary digits in fours, starting at the least significant (right-hand side of the number) and

Hex	Binary	Hex	Binary
Ø	ØØØØ	8	1ØØØ
1	ØØØ1	9	1ØØ1
2	ØØ1Ø	A	1Ø1Ø
3	ØØ11	B	1Ø11
4	Ø1ØØ	C	11ØØ
5	Ø1Ø1	D	11Ø1
6	Ø11Ø	E	111Ø
7	Ø111	F	1111

Fig. 3.2. Hex and binary digits.

Conversion: Hex to Binary

Example: 2CH ..... 2H is ØØ1Ø binary
CH is 11ØØ binary

So 2CH is ØØ1Ø11ØØ binary (data byte)

Example: 4A7FH ..... 4H is Ø1ØØ binary
AH is 1Ø1Ø binary
7H is Ø111 binary
FH is 1111 binary

So 4A7FH is Ø1ØØ1Ø1ØØ1111111 binary (an address)

Conversion: Binary to Hex

Example: Ø11Ø1Ø11 ..... Ø11Ø is 6H
1Ø11 is BH

So Ø11Ø1Ø11 is 6BH

Example: 1Ø11Ø1ØØ1ØØ1Ø ... note that this is not a complete number of bytes.

Group into fours, starting with lsb:
ØØ1Ø is 2H
1ØØ1 is 9H
11Ø1 is DH and
the remaining 1Ø is 2, making 2D92H

Fig. 3.3. Converting between hex and binary.

then convert each group into its corresponding hex digit. Fig. 3.3 shows examples of the conversion in each direction so that you can see how easy it is.

The Dragon is one of the better designed computers in as much as it can convert between hex and denary by means of BASIC commands. If you want to find what a denary number looks like in hex, just use HEX$(number). For example, ?HEX$(46) will give on the screen 2E, which is the hex for 46 denary. To find the denary equivalent of hex, use &H. For example, ?&HCØØ4 will give you 49156 on the screen. The provision of this built-in converter means that you don't have to bother with learning the rather tedious methods for converting between denary and hex. Just in case you want to do this when you are not near a friendly Dragon, though, the methods are shown in Appendix C. The Dragon, however, has no built-in assembler. An assembler would allow you to write machine code almost as easily as you write BASIC, by using command words. Since an excellent assembler is available on cartridge (Appendix B) however, this is not a catastrophic loss.

Assuming, which is reasonable, that you don't want to commit yourself to the cost of a full-scale assembler at this point, what do you do to create machine code programs? The answer is that you should design your program in assembly language, which is by far the easiest way to design machine code programs, and then you should convert into hex code. Converting means looking up – in a set of tables called the *instruction set*– the hex number that represents each instruction. Instruction sets are provided by the manufacturers of all microprocessors, and Motorola, who designed the 6809, provide one for this chip. Just to assist you, a quick reference guide has been included in this book, in Appendix D. Don't refer to it at the moment – it'll put you off!

Negative numbers

Useful as the denary-hex conversion commands of the Dragon are, they are deficient in one respect – they don't handle negative numbers. Now this is unfortunate, though understandable. Negative numbers are very important in machine code programs, particularly if you are working without an assembler. The reason is that you sometimes want the MPU to do the equivalent of a GOTO, perhaps jumping to a step which is thirty steps ahead of its present address. This sort of thing is usually programmed by supplying a data

number which is the number of steps that you want to skip. If you want to jump back to a previous step, however, you will need to use a negative number for this data byte. This is very common, because it's the way that a loop is programmed in machine code. We need, therefore, to know how to write a negative number in hex.

What makes it awkward is that there is no negative sign in hex arithmetic. There isn't one in binary, either. The conversion of a number to its negative form is done by a method called *complementing*, and Fig. 3.4 shows how this is done. At first sight, and very often at second, third, and fourth, it looks entirely crazy. When you are dealing with a single byte number, for example, the denary form of the number −1 is 255! You are using a large number

```
Binary: Number (8 bits)        ØØ11Ø1Ø1 (denary 53)

Step 1: change each Ø to
        1, each 1 to Ø         11ØØ1Ø1Ø

Step 2: add 1                        +1
                               ---------
Result is negative
form of number                 11ØØ1Ø11 (denary 2Ø3)
                               =========

Denary: Number                       53

Step 1: subtract from 256            2Ø3

This is negative form (in denary)

Hex: Number                          $35     Remember that F
                                             represents 15
Step 1: subtract from $FF            FF      denary, and
                                     35      15 - 5 = 1Ø
                                     ---     denary, which
                                     $CA     is A in hex.

Step 2: add 1                        +1      This is easier
                                     ---     than a subtraction
                                     $CB     from $1ØØ, which
                                             involves the use
                                             of carries in hex.

Result is hex form of negative number

An alternative is to do the denary conversion, then
convert back to hex.
```

Fig. 3.4. The two's complement, or negative form, of a binary number.

to represent a small negative one! It begins to make more sense when you look at the numbers written in binary. The numbers that can be regarded as negative all start with a 1 and the positive numbers all start with a Ø. The MPU can find out which is which just by testing the left-hand bit, the most significant bit.

It's a simple method, which the machine can use efficiently, but it does have disadvantages for humans. One of these disadvantages is that the digits of a negative number are not the same as those of a positive number. For example, in denary −4Ø uses the same digits as +4Ø. In hex, −4Ø becomes $D8 and +4Ø becomes $28. The denary number −85 becomes $AB and +85 becomes $55. The second disadvantage is that humans cannot distinguish between a single byte number which is negative and one which is greater than 127. For example, does $9F mean 159 or does it mean −97? The short answer is that the human operator doesn't have to worry. The microprocessor will use the number correctly no matter how we happen to think of it. The snag is that we have to know what this correct use is in each case. Throughout this book, and in others that deal with machine code programming, you will see the words 'signed' and 'unsigned' used. A *signed number* is one that may be negative or positive. For a single byte number, values of Ø to $7F are positive, and values of $8Ø to $FF are negative. This corresponds to denary numbers Ø to 127 for positive values and 128 to 255 for negative. *Unsigned numbers* are always taken as positive. If you find the number $9C described as signed, then, you know it's treated as a negative number (it's more than $8Ø). If it's described as unsigned, then it's positive, and its value is obtained simply by converting. How do we convert a signed single-byte hex number into denary, when the Dragon won't accept negative numbers for conversion? Simple, just type ?&HXX−256, where XX is the hex number. For example, ?&HFF−256 will give you −1, which is the correct value for $FF treated as a negative number.

Light relief

Just to take a break from all this arithmetic (that's what they called it before it became known as 'maths'!), let's look at screen displays on the Dragon. Each part of the screen can be controlled by whatever is stored in part of the memory, but there are two varieties of this memory. The simplest one to work with is the part which is called 'text screen memory'. It takes up memory addresses $4ØØ to $5FF,

which is 1Ø24 to 1535 in denary. Notice, incidentally, how neatly the range fits in hex scale. What we mean by 'text screen memory' is that this piece of memory is treated in a special way. Anything that is stored here will be used to display text on the screen. That means that any ASCII code that you store at an address in this range will produce the corresponding letter or graphics shape on the screen. Try this – press the CLEAR key to clear the screen, and then type:

POKE&H5ØØ,65

and press ENTER. The result is the letter A appearing at the left-hand side of the screen, halfway down. The computer has converted the ASCII code of 65 that was stored at position &H5ØØ into a set of numbers that will produce the letter 'A' on the screen at this position. You can program this sort of thing in a loop, as Fig. 3.5 shows. You

```
10 FOR N=&H400 TO &H5FF
20 POKE N,65:NEXT
```

Fig. 3.5. A program which fills the screen with A's.

have to be careful how you type this. If you don't leave a space between the last Ø of &H4ØØ and the 'T' of TO, then the computer tries to read the T as being part of the hex number, and stops with an SN ERROR message. Look out for this sort of thing when you make use of hex numbers in POKE instructions.

The effect of this loop is to fill the screen with A's. The filling is not particularly fast, because we're using BASIC in the loop. Later, we'll look at the same sort of thing in machine code, which is stunningly fast! For the moment, though, look at the effect on this program of replacing 65 by another number which is the code for one of the graphics blocks, like 136. It's quite useful and, if you want to avoid the 'hole' in the pattern caused by the OK message and the cursor, then add an endless loop to the program, like:

3Ø GOTO 3Ø

If you want to see this zip along a lot faster, then turn on the high speed clock rate with POKE&HFFD7,Ø. This is the same command as we used before, but in hex terms instead of denary.

The Dragon uses memory for graphics as well, however, and the memory for this is organised in a different way. The graphics

```
10 PMODE0:PCLS:POKE&H700,65
20 SCREEN1,0
100 GOTO100
```

Fig. 3.6. Poking to the graphics memory

memory is arranged in eight pages, each of $6ØØ bytes. This is a lot more than the $2ØØ bytes that we use for the text screen, and its use is not so automatic. If we poke a byte into part of this memory, for example, we'll see nothing on the screen. This is because the Dragon normally displays the character that corresponds to a byte in the text screen memory. To see what is stored in the graphics memory, we have to switch to a graphics mode, and select the correct page. Try, for example, the program in Fig. 3.6. This selects a mode, in this case PMODEØ, and then clears the graphics screen memory, using PCLS, and then pokes the number 65 into address $7ØØ. This is an address in graphics page 1, but it can't produce any effect on the text screen. To make it visible, we have to switch to displaying a graphics mode. Using PMODEØ makes use of page 1 for display, and SCREEN1,Ø makes the effect visible. The result – well, take a look for yourself! It's a couple of bright spots near the top left-hand corner.

The next thing we have to figure out is why the number 65 (denary) should cause these two dots to appear. To help in this investigation, alter the program so that the number 255 is poked, and try it again. This time you see a line at the left-hand side. Now try poking the number &H55. This gives four dots. Light dawns when we write these numbers in binary form. 65 denary is $41, and in binary it's ØIØØØØØI. The number 255 is IIIIIIII, and $55 is ØIØIØIØI. Where there's a 1 in the binary number, there's a dot on the screen! The memory address &H7ØØ must service this strip of the screen. Now try the same program using the other PMODE numbers, 1 to 4. In the modes Ø,2, and 4 you will see the pattern appear as light on dark. In modes 1 and 3, it appears as white spots on green. The spacing between the spots also changes from one mode to another. The greater the resolution of the graphics mode, the closer and smaller are the spots. If you aren't too confident about graphics modes, then you should brush up with the chapter in my book, *The Dragon 32 and How to Make the Most of It*, or take a look at the more advanced book by Steve Money, *Dragon Graphics and Sound*.

We can use this method to create patterns of our own. If we stick

```
10 PMODE2:PCLS:POKE&H700,24
20 FOR N=1TO7:READ D:POKE&H700+N*16,D
30 NEXT:SCREEN 1,0
40 GOTO40
100 DATA24,255,255,170,170,170,170
```

Fig. 3.7. A program which creates a shape on the graphics screen.

to PMODE0 at the moment, then each line on the screen starts at an address which is $10 greater than the one before. For example, if you add the line:

```
15 POKE&H710,255
```

to the program of Fig. 3.6, you'll see a line appear under the row of spots. We can, therefore, create shapes on the screen by poking numbers to addresses that are $10 apart in successive lines. This is illustrated in the program of Fig. 3.7. When you RUN this one, you will see an octopus (well, quadropus) shape appearing. This has been programmed by poking numbers into the graphics screen memory. How did we get the numbers? Figure 3.8 shows how. I used an

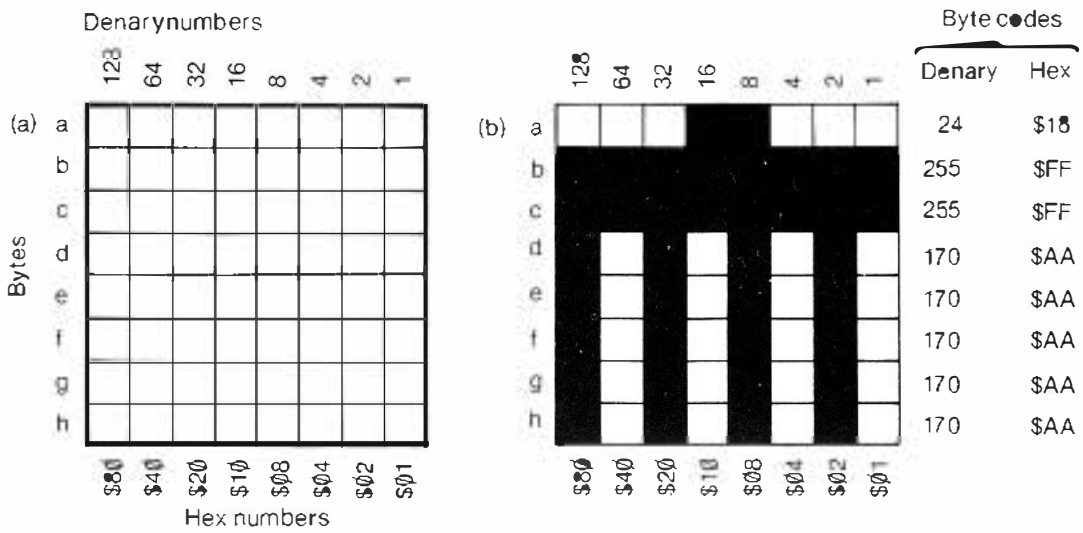


Fig. 3.8. Planning shapes: (a) the planning grid, (b) the example which has been used. The grid is of 8 × 8 squares, but larger number of lines of eight squares across could be used if wanted.

arrangement of 8 × 8 squares, and shaded in the ones that I wanted to see lit. Since I'm using the eight squares across to represent positions in a binary number, I have written in the denary values that each square represents. By adding these up for each line, a set of numbers is found. For example, the 'head' in the first line has squares 16 and 8 shaded, totalling 24. This is the number that is poked into the first of the set of addresses. The next two form the body, using all squares lit, number 255. There's no reason to confine ourselves to just eight squares down – we can use as many as the screen will hold. We have to confine ourselves to 8 across at any one memory address, however, because there are only eight bits in a byte. It opens up a whole new set of prospects in graphics, doesn't it?

Chapter Four
6809 Details

Registers - PC and accumulators

A microprocessor consists of sets of memories, of a rather different type to ROM or RAM, which are called *registers*. These registers are connected to each other and to the pins on the body of the MPU by the circuits that are called *gates*. In this chapter, we shall look at some of the most important registers of the 6809 and how they are used. A good starting point is the register which is called the PC - short for *Program Counter*.

No, it doesn't count programs - what it does is to count the steps in a program. The PC is a sixteen-bit (two byte) register which can store a full-sized address number, up to $FFFF (65535 denary). It's purpose is to count the address number, and the number that is stored in the PC will be incremented (increased by 1) each time an instruction is completed, or when another byte is needed. For example, if the PC holds the address $1F3A, and this address contains an instruction byte, then the PC will increment to $1F3B whenever the MPU is ready for another byte. The next byte will then be read from this new address.

What makes the PC so important is that it's the automatic way by which the memory is used. When the PC contains an address number, the electrical signals that correspond to the Ø's and 1's of that address appear on a set of connections, collectively called the *address bus*, which link the MPU to all of the memory, RAM and ROM. The number that is stored in the PC will select one byte from the memory, the byte which is stored at that address number. At the start of a read operation, the MPU will send out a signal called the *read signal* on another line, and this will cause the memory to connect up the selected parts to another set of lines, the *data bus*. The signals on the data bus then correspond to the pattern of Ø's and 1's that is stored in the byte of memory that has been selected by the

address in the PC. Each time the number in the PC changes, another byte of memory is selected, so that this is the way by which the MPU can keep itself fed with bytes. When the MPU is ready for another byte, the PC increments, and another read signal is sent out.

There are other ways in which the PC number can be changed, but for the moment we'll pass over that and look at another register, the *accumulator*. The accumulator of a microprocessor is the main 'doing' register of the MPU. This means that you would normally use it to store any number that you wanted to transfer somewhere else, or add to or carry out any other operation upon. The name of accumulator comes from the way in which this register operates. If you have a number stored in the accumulator, and you add another number to it, then the result is also stored in the accumulator. The nearest equivalent in BASIC is using a variable A, and writing the line:

```
A=A+N
```

where N is a number variable. The result of this BASIC line is to add N to the old value of A, and make A equal to this new value. The old value of A is then lost. The accumulator acts in the same way, with the difference that an accumulator can't store a number greater than 255 (denary).

The 6809 has two accumulator registers, labelled A and B. The importance of these is that they are used much more than the other registers. This is because so many actions can be carried out more quickly, more conveniently, or perhaps only, in such an accumulator. When we read a byte from the memory, we usually place it in one of these accumulators. When we carry out any arithmetic or logic action, it will normally be done in an accumulator and the result also stored in the accumulator.

Addressing methods

When we program in BASIC, we don't have to worry about memory addresses at all unless we are using PEEK or POKE. The task of finding where bytes are stored is dealt with by the operating system of the machine. When a variable is allocated a value in a BASIC program, as, for example, by a line like:

```
10 N=12
```

we never have to worry about where the number 12 is stored, or in what form. Similarly, when we add the line:

```
2Ø K=N
```

we don't have to worry about where the value of N was stored or where we will store the value of K. Remembering our comparison with wall building, we can expect that when we carry out machine code programming, we shall have to specify each number that we use, or alternatively the address at which the number is stored. This way in which we obtain a number, or find a place to store it, is called the *addressing method*. What makes the choice of addressing method particularly important is that a different code number is needed for each different addressing method for each command. This means that each command exists in several different versions, with a different code for each addressing method. A list of all the 6809 addressing methods at this stage would be rather baffling, and for that reason has been consigned to Appendix E. What we shall do here is to look at some examples of selected addressing methods and the way that we write them in assembly language.

Assembly language

Trying to write down machine code directly as a set of numbers is a very difficult process which is liable to errors from beginning to end. The most useful way of starting to write a program is to write it in a set of steps in what is called *assembly language* (or *assembler language*). This is a set of abbreviated command words, called *mnemonics*, and numbers which are the data or address numbers. The numbers can be in hex or in denary, provided they are supplied to the computer in the correct form. Each line of an assembly language program indicates one microprocessor action, and this set of instructions is later 'assembled' into machine code, hence the name.

The aim of each line of an assembly language program is to show the action and the data or address that is needed to carry out that action, just as when we make use of TAB in BASIC we need to complete the command with a number. The part of the assembly language that specifies what is to be done is called the *operator*, and the part which specifies what the action is done to or on is called the *operand*. A few instructions need no operand, and we'll look at some later.

An example makes this easier. Suppose we look at the assembly language line:

```
LDA #$12
```

The operator is LDA, a shortened version of LOAD A, meaning that the accumulator register A is to be loaded with a byte. The operand is #$12, of which the $12 means that this is 12 hexadecimal, rather than twelve denary. The other mark, the hashmark #, is used to show the addressing method that is to be used, a method called *immediate addressing*.

The whole line, then, should have the effect of placing the number $12 into the accumulator register A. It is the equivalent in machine code terms of the BASIC instruction:

```
A=&H12
```

You could imagine that the memory which held the number was inside the microprocessor rather than part of the RAM memory, and was labelled with the name A.

A command such as LDA #$12 is said to use immediate addressing, because the byte which is loaded into the accumulator must be placed in the memory byte whose address immediately follows that of the instruction byte. There is one code number for the LDA # part of the whole instruction, and this byte is $86, so that the hex sequence in memory of 86 12 will represent the entire command LDA #$12. It's a lot easier to remember what LDA #$12 means than to interpret 86 12, however, which is why we use assembly language as much as possible.

Immediate addressing like this can be convenient, but it ties you down to the use of one definite number. It's rather like programming in BASIC:

```
N=4*12 + 3
```

rather than

```
N=A*B+C
```

In the first example, N can never be anything else but 51, and we might just as well have written: N = 51. The second example is very much more flexible, and the value of N depends on what values we choose for the variables A, B and C. When a machine code program is held in RAM, then the numbers which are loaded by this immediate addressing method can be changed if we must change

them, but when the program is held in ROM no change is possible. That's just one reason for needing other addressing methods. One of these other methods is *extended addressing*.

Extended addressing uses a complete two-byte address as its operand. This creates a lot of work for the 6809 because, when it has read the code for the operator, it will then have to read two more bytes to find the memory address at which the data is stored. It will then have to place this address in the PC, read in the data byte, carry out the operation, and then restore the next correct address into the PC. Fig. 4.1 shows in diagram form what has to be done. An

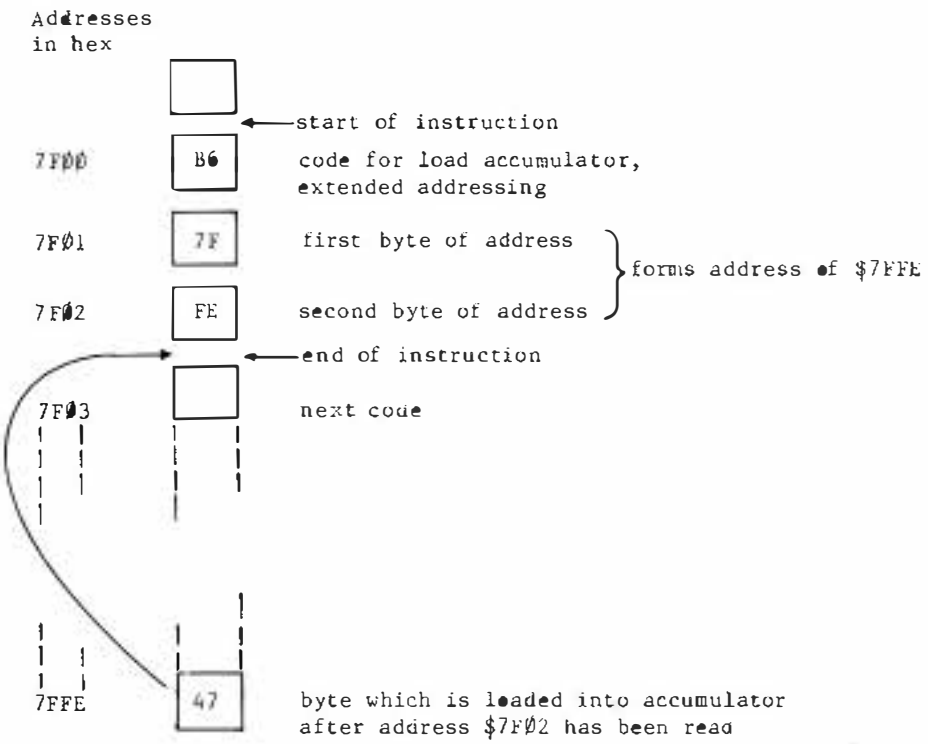


Fig. 4.1. How the extended addressing method works.

extended-addressed operation is therefore a lot slower to carry out than an immediate one. It's easy to alter the data, however, since any byte may be stored at the address which is specified.

Suppose, for example, that we have the instruction:

```
LDA $7FFE
```

In this slice of assembly language, the operator is LDA (load the accumulator A) and the operand is the address $7FFE. What you have to remember is that what is put into the register A is not 7FFE, which is a two-byte address, but the data byte which is stored in

memory at this address. The effect of the complete instruction, then, is to place a copy of the byte which is stored at $7FFE into the accumulator A of the 6809. When the instruction has been completed, the address $7FFE will still hold its own copy of the byte, because reading a memory does not change the content of the memory in any way.

We can also use the extended addressing method in a command which will store a byte into the memory. The command:

```
STA $7FFF
```

means that the byte that is stored in the accumulator A is to be copied to memory at address $7FFF. This action does change the content of this memory address, but the accumulator A will still hold the same byte after the instruction has been carried out.

Direct page addressing

Direct page addressing is a method that allows you to specify a full address by using only one byte! The secret is another register, the Direct Page (DP) register. This can be loaded with the upper byte of an address number by transferring a byte from another register. When, later in the program, an address is needed, then only the lower byte needs to be specified. The byte in the DP register will be taken as being the upper byte of the address. For example, if most of the important bytes that a program might need were stored in the range $5FØØ to $5FFF, then we could load $5F into the DP register. A step such as:

```
LDA $3F
```

would then load the accumulator with a copy of the byte that is stored at address $5F3F. The $5F has been provided from the DP register, and the $3F from the data byte that followed the LDA instruction. When only one byte is specified following an instruction like LDA, then an assembler will automatically use the code for DP addressing. The DP register of the Dragon is usually set to $ØØ.

Indexed addressing

Indexed addressing is a method which is particularly useful on the 6809. The principle is that a sixteen-bit register is used to hold an

address, the address of a byte in the memory. This address can be used directly, or it can be used as a base address. Use as a base address means that we can add a number to the address before using it. For example, suppose that we had the number $7000 stored in an index register. If we like, we can use this to specify the use of the address $7000, or alternatively we can add a number to it. If we add 4, for example, then we could load from $7004 or store to this number.

There are five registers in the 6809 that can be used in this way, the X,Y,U,S. and PC registers, but we generally make most use of the X and Y registers in this type of addressing. Once we have decided which register to use, there are still several choices of the way in which a number can be added to the address. The simplest method is called *constant indexed addressing*. A number constant is specified in the command, and this number is added to the number in the register before it is used as an address. For example, the line:

```
LDA 5,X
```

means that there is an address stored in the X register, and 5 will be added to this address number before it is used. We can also, in assembly language, use a label, a sort of 'variable name' for a number. We could, therefore, have:

```
LDA CONST,X
```

where CONST means a number that has been assigned to the label name of CONST. As before, this number will be added to the number stored in the X register, and the result used as an address. A useful feature of the 6809 is that this number or label which is added in can be zero. It can also be a signed number of 5, 8 or 16 bits, which allows the range of memory that this addressing method can cover to be very large. It's important to realise that when you make use of an index register in this way, the number that is stored in the index register is not changed – the result of the addition is not put back into the register.

There are also more exotic variations on indexed addressing. One of them is *accumulator indexed addressing*. In this variety, the number in one of the accumulators (A, B or D) is added to the number in the index register before the total is taken. The D accumulator is not really another different register, it's simply the A and B registers combined and used as one sixteen-bit register. A more interesting variety for us at the moment is the method that is described as *auto increment/decrement zero offset indexed*. That

mouthful means simply that we can increment or decrement the number in the index register each time it is used. When this method is used, the number that is added to the register content is zero (hence 'zero-offset'), and we specify the increment by a + sign, decrement by a − sign. We can also specify one increment or two ; adding + will increment once, adding ++ will increment twice, and the same idea applies to the use of the − signs. Using these commands does change the number that is stored in the index register.

This auto increment/decrement method allows us to load from or store to a different address each time the instruction is used. The way in which the instruction is carried out is rather special, though. When we use auto increment, the incrementing of the register is carried out after the operation. For example, if we write the command:

```
LDA Ø,X+
```

then the accumulator A will be loaded from the address that is stored in the X register. After the loading is complete, the number in the address register will be incremented once. Putting the + sign in the assembly language command after the X reminds us that the increment action is carried out after the loading (or whatever) operation. When a decrement is used, the register number is decremented before the operation, and the− sign is placed before the register name as a reminder. Take, for example:

```
STA Ø,--X
```

This means that the number in the X register will be decremented twice (two subtracted), and then used as the address for storing the byte in the A register.

Indirect addressing

Indirect addressing means going to an address to pick up another address at which a byte is located. It's like going to the address of a tourist office to find the address of a hotel (for a quick byte?). The 6809 allows several of its addressing methods to be used in this way. which seems rather exotic at this stage but turns out to be very useful when you get deeper into programming. Indirect addressing is indicated in assembly language by the use of square brackets, and it

can be used with extended addressing and all the forms of indexed addressing.

Just to give one brief example, the line:

```
LDA [$7FFE]
```

means indirect extended addressing. The data byte that we want is *not* at the address $7FFE. Instead, its high byte is at the address $7FFE and its low byte at $7FFF. If we imagine that $3A is stored at $7FFE, and $47 is stored at $7FFF, then the address that the 6809 will actually use to load from is the address $3A47. What's useful about this? Well, just the point that the bytes in the addresses $7FFE and $7FFF can be changed, even within the program, to make the loading come from different addresses. It all, believe it or not, makes the task of the programmer easier, and seasoned programmers who are not tied to the continual use of other types of microprocessors are rather keen on the many different ways that the 6809 allows you to address memory.

Relative addressing

Relative addressing is one of the first addressing methods that was ever used, and it is not used for many commands nowadays. Relative addressing means that the operand of an instruction can be one or two bytes, and the address that is going to be used is found by adding this number (called the 'offset') to the 'current address', which is the number in the program counter. It's rather like the old-style Treasure Island maps which specify 'one step left, two forward, three right ...' and so on. You don't know where this will get you until you know where to start, but when relative addressing is used in a microprocessor, the starting place is usually the address in the PC.

The 6809 uses relative addressing for its BRANCH commands. There are two varieties, short and long. The short branches use a single-byte number, signed, as an offset. The use of a single-byte signed number means that we can jump to a new address which is up to 127 steps forward or 128 steps back from the present one. The alternative is long-relative branching. This uses a two-byte number, treated as signed, following the command, and it allows a range of +32767 to −32768 steps from the address of the branch command. A long branch can be to any part of the normal 64K memory, but its use takes more time than the short branch. These branches are the machine code equivalents of GOTO, but with the difference that

they can be made to depend on a condition, like the accumulator containing zero. It's as if there were one single BASIC instruction which carried out the effect of:

IF A=Ø THEN GOTO...

We'll look at branch instructions in a lot more detail later.

The other registers

Of the other registers, S and U are both sixteen-bit registers that we shall leave strictly alone during the course of this book, until we get to Chapter 9. They are the type of registers that are called *stack pointers*, and they are used to locate bytes which the MPU has stored temporarily. If you interfere with what is stored in the S register, you may upset the operating system of the computer. The U register is safer to use, but in this book we shall not be concerned with it. The Direct Page register DP is used in connection with direct page addressing, as already mentioned. That leaves the CC (Condition Code) register to deal with in more detail now.

The CC register

The *condition code register*, sometimes called the Flag or Status register, isn't really a register like the others. You can't do anything with the bits in this register, and they don't even fit together as a number. What the CC register is used for is as a sort of electronic

```
Number in accumulator    1Ø11ØØ11Ø
Number added             11ØØØ1Ø1
                         ---------
Result                  1Ø1111Ø11

This consists of nine bits, and the accumulator
can hold only eight.  The most significant bit
is transferred to the carry flag of the status
register.

Accumulator now holds Ø1111Ø11
Carry bit is set (equal to 1)
```

Fig. 4.2. Why the carry bit is needed.

note-pad. Each bit in the register (there are eight of them) is used to record what happened at the previous step of the program. If the previous step was a subtraction that left the A register storing zero, then one of the bits in the CC register will go from value ∅ to value 1 to bring this to the attention of the MPU. If you add a number to the number in an accumulator, and the result consists of nine bits instead of eight (Fig. 4.2) then another of the bits in the CC register is 'set', meaning that it goes from ∅ to 1. If the most significant bit in a register goes from ∅ to 1 (which might mean a negative number), then another of the CC bits is set. Each bit, then, is used to keep a track of what has just happened. What makes this register important is that you can make branch commands depend on whether a CC bit is set (to 1) or reset (to ∅).

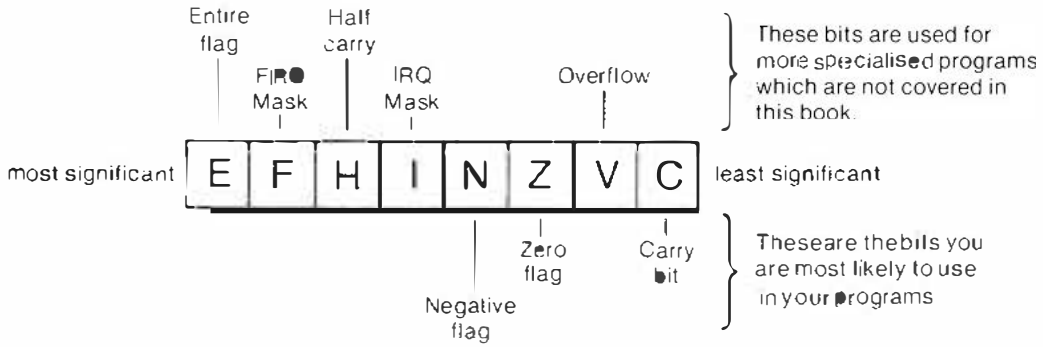


Fig. 4.3. The bits of the CC register. Only three of these are extensively used in most programs.

Figure 4.3 shows how the bits of the CC register of the 6809 are arranged. Of these bits, ∅,2 and 3 are the ones that we are most likely to use at the start of a machine code career. The use of the others is rather more specialised than we need at the moment. Bit ∅ is the Carry Flag or Bit. This is set (to 1) if a piece of addition has resulted in a carry from the most significant bit of a register. If there is no carry, the bit remains reset. When a subtraction is being carried out (or a similar operation like comparison), then this bit will be used to indicate if a 'borrow' has been needed. It can for some purposes be used as a ninth bit for either accumulator, particularly for shift and rotate operations in which the bits in a byte are all shifted by one place (Fig. 4.4.).

The Zero Flag is bit 2. It is set if the result of the previous operation was exactly zero, but will be reset (∅) otherwise. It's a useful way of detecting equality of two bytes – subtract one from the other and, if the Zero Flag is set, then the two were equal. The Negative Flag is set if the number resulting in a register after an

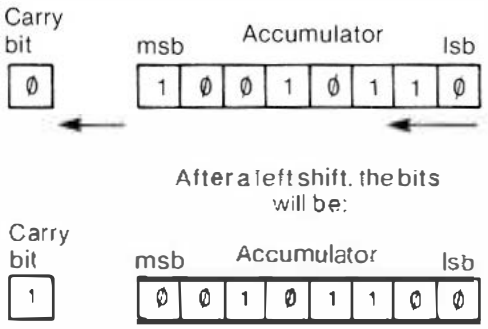


Fig. 4.4. Using the Carry Bit (or Flag) in a shift operation, in which all the bits of a byte are shifted one place to the left.

operation has its most significant bit equal to 1. This is the type of number that might be a negative number if we are working with signed numbers. This bit is therefore used extensively when we are working with signed numbers. Unlike most MPU's, the 6809 allows the programmer to work with the contents of the CC register. You can transfer its contents (which means copy – the bits are not moved out of the CC register) to any other single byte register. More usefully, you can exchange the contents of the CC register with the contents of any other single byte register. This will change the contents of the CC register. There are also a few commands which allow you to change some of the bits of the CC register selectively – but that's not beginner's work! Keep it in mind for when you're an expert.

Chapter Five

Register Actions

Accumulator actions

Since the accumulator is the main single-byte register, we can list its actions and describe them in detail, knowing that whatever holds good for the 'A' accumulator of the 6809 will also hold good for the 'B' accumulator (with one minor exception). Of all the accumulator actions, simple transfer of a byte is by far the most important. We don't, for example, carry out any form of arithmetic on ASCII code numbers, so that the main actions that we perform on these bytes are loading and storing. We load the accumulator with a byte copied from one memory address, and store it at another. Very few computer systems allow a byte to be moved directly from one address to another, so that the rather clumsy-looking method of loading from one address and storing to another is used almost exclusively.

The next most important group of actions is the arithmetic and logic group, which contains addition, subtraction, AND, OR and COM (the logical NOT action). The multiply action also falls into this group, and we can add to it the SHIFT and ROTATE actions which we looked at briefly in the previous chapter. The effects of the 6809's shift and rotate commands, with their assembly language mnemonics, are shown in Fig. 5.1. A shift always results in a register losing one of its stored bits, the one at the end which is shifted out. Most types of shifts – the arithmetic shift right is the exception – cause the register to gain a zero at the opposite end. The carry bit is used as ninth bit of the accumulator in all of these shifts. The shift action can be carried out on either the 'A' or the 'B' accumulator, or on a byte that is stored in the memory. The effect of a shift on a binary number stored in the register is to multiply the number by two if the shift is left, or to divide it by two if the shift is right (Fig. 5.2).

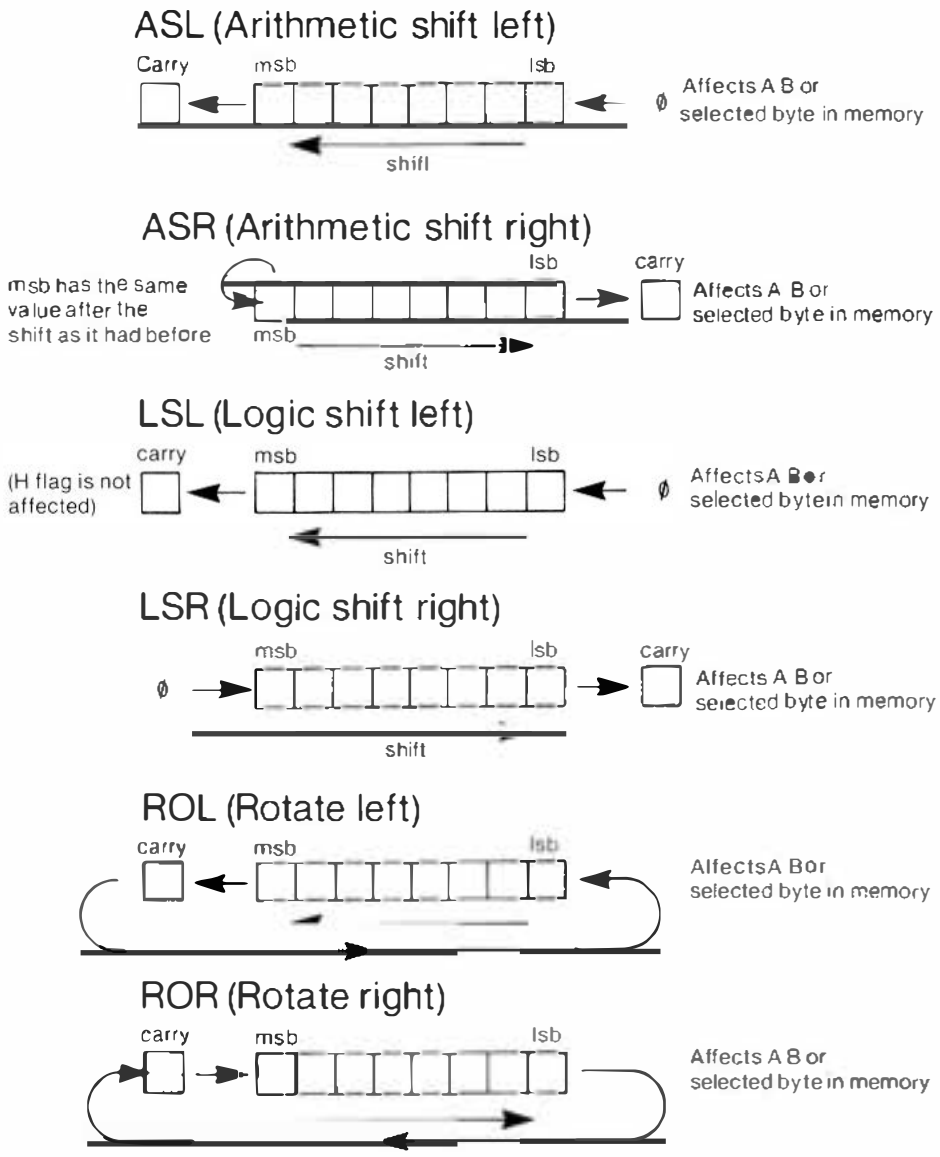


Fig. 5.1. The 6809 shift and rotate instructions.

A rotation, by contrast, always keeps the same bits stored in the register, but the positions of the bits are changed. The 6809 has two rotate commands, one for rotate right and the other for rotate left. Once again, they use the carry bit as the ninth bit of the register. Either A or B accumulator can be used, and the action can be carried out on a byte stored in the memory.

A third group of accumulator actions contains the increment, decrement and comparison commands. Increment means adding 1 and decrement means subtracting 1. The command whose mnemonic is INCA will therefore have the effect of adding 1 to the

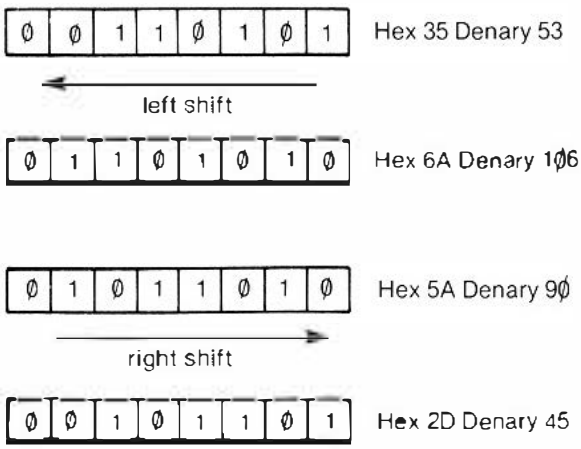


Fig. 5.2. The effect of a shift on a number.

number that was stored in accumulator register A. Similarly, DECB will have the effect of subtracting 1 from the number that was stored in accumulator B. These actions can also be carried out on any byte that is stored in the memory. Their actions will affect the zero and negative flags in the CC register, so that if either command causes the register or memory byte to contain zero, or to contain a number whose most significant bit is set, then the flags reveal this.

The CMP (Compare) instruction is a particularly useful one. Looking at the moment at its use applied to the 'A' accumulator only, CMPA is the mnemonic. It must use one of the standard memory addressing methods, and its effect is to compare the byte that was copied from the memory with the byte that is already present and stored in accumulator A. Compare in this respect means that the byte copied from memory is subtracted from the byte in the accumulator. The difference between this instruction and a true subtraction is that the result is not stored anywhere! The result of the subtraction is used to set flags, but nothing else, and the byte in the accumulator is unchanged. For example, suppose that the accumulator contained the byte $4F, and we happen to have the same size of byte stored at address $327F. If we use the command:

```
CMPA $327F
```

then the zero flag in the CC register will be set (to 1), but the byte in the accumulator will still be $4F, and the byte in the memory will still be $4F. A subtraction would have left the content of the accumulator equal to zero.

Why should this be important? Well, suppose you want a program to do one thing if the 'Y' key is pressed, and something different if the 'N' key is pressed. If you arrange for the machine code program to

store into the accumulator the ASCII code for the key that was pressed, you can compare it. By comparing it with $4E (the ASCII code for 'N'), we can find if the 'N' key was pressed. If it was, the zero flag will be set. If not, we can test again. By comparing with $59, we can find if the 'Y' key was pressed – once again, this would cause the zero flag to be set. If neither of these comparisons caused the zero flag to be set, we know that neither the 'Y' nor the 'N' key was pressed, and we can go back and try again. If it looks very much like the action of the INKEY$ loop in BASIC, you're right – it is.

Finally, we have the test-and-branch actions. These, as the name suggests, allow the flags in the CC register to be tested, and will make the program branch to a new address if a flag was set. Which flag? That depends which branch-and-test instruction you use, because there's a different one for each flag, and for each state of a flag. For example, consider the two tests whose mnemonics are BEQ and BNE. BEQ means 'branch if equal to zero'. As this suggests, it will cause a branch if the result of a subtraction or comparison is zero. In other words, it causes a branch to take place if the zero flag is set. It's 'opposite number' BNE means 'branch if not equal to zero'. It will cause a branch to take place if the zero flag is not set. There are, therefore, two branch instructions which test the zero flag, but in opposite ways. The same sort of thing goes for most of the other flags. There's also a branch instruction, mnemonic BRA, which doesn't carry out any tests, like a GOTO with no IF preceding it, and a curious 'branch never' which doesn't do anything! In fact, it's useful for wasting time or leaving room for code you haven't written yet.

The complete list of all the available branch instructions is shown in Fig. 5.3. Many of these are instructions that you'll probably never use, and the really important ones are the ones that use the zero, carry and negative flags. All of them use relative addressing. There are two varieties of relative addressing used with these instructions. The ordinary variety needs one number byte to follow the code for the branch. This number, as we noted earlier, is treated as a signed number (in other words, if it's more than $7F it's treated as negative). It is added to the address which is in the PC at the instant when the branch is carried out. The result of this addition is the address to which the branch goes, so the next instruction that is carried out will be the one at this address. This type of branch, which uses a single byte 'displacement' number, permits a shift of up to 127 (denary) places forward, or 128 backward. That's because a single signed byte can't exceed these values. The 6809 also permits what is

	Mnemonic	Meaning	Code	Effect
Unconditional:	BRA	Branch always	$2Ø	causes a branch irrespective of previous action
	BRN	Branch never	$21	wastes time
	BSR	Branch to subroutine	$8D	go to subroutine
Conditional, simple	BMI	Branch on negative	$2B	branch if N flag is set (1)
	BPL	Branch on positive	$2A	branch if N flag is reset (Ø)
	BEQ	Branch on zero	$27	branch if Z flag is set (1)
	BNE	Branch on not-zero	$26	branch if Z flag is reset (Ø)
	BVS	Branch on overflow	$29	branch if V flag is set (1)
	BVC	Branch on no overflow	$28	branch if V flag is reset (Ø)
	BCS	Branch on carry set	$25	branch if C flag is set (1)
	BCC	Branch on carry clear	$24	branch if C flag is reset (Ø)
Conditional, signed numbers	BGT	Branch on greater	$2E	branch if comparison positive
	BLE	Branch on lesser	$2F	branch if comparison negative
	BGE	Branch greater/equal	$2C	branch if comparison positive or equal
	BLT	Branch lesser/equal	$2D	branch if comparison negative or equal
Conditional, unsigned numbers	BHI	Branch on greater	$22	branch if comparison positive (unsigned)
	BLS	Branch on lesser	$23	branch if comparison negative (unsigned)
	BHS	Branch on greater/equal	$24	branch if comparison positive or zero (unsigned)
	BLO	Branch on lesser/equal	$25	branch if comparison negative or zero (unsigned)

Note: Long branches use L in front of mnemonic (LBRA, LBEQ) and use an extra $1Ø byte in front of the branch byte (1Ø 2Ø or 1Ø 27)

Fig. 5.3. The complete list of 6809 BRANCH instructions.

called a 'long branch', with a two-byte number following the instructions code. This allows a branch of up to 32767 (denary) places forward or up to 32768 places backward.

Interacting with Dragon

The time has come now to start some practical machine code programming of your Dragon. This is not simply a matter of typing the assembly language lines as if they were lines of BASIC. Unless you happen to have an assembler program cartridge fitted, the Dragon will simply give you 'SN ERROR' messages when you try to run these programs. Since we want to start on a small scale, we'll forget about assemblers at the moment, and assemble 'by hand'. This means that we find the machine code bytes that correspond to the assembly language instructions by looking them up in a table. We then poke them into the memory of the Dragon, place the address of the first byte into the PC of the 6809, and watch it all happen. It sounds simple, but there is quite a lot to think about, and a number of precautions to take. To start with, the Dragon uses quite a lot of its RAM, as we have seen, for its own purposes. If we simply POKE a number of bytes into the memory without heeding

which part of memory we use, the chances are that we shall either replace bytes that the Dragon needs to use, or our program bytes will be replaced by the action of the Dragon. What we need is a piece of memory that is safely roped off for our use only.

This can be done by making use of the CLEAR command in BASIC. When you use CLEAR in the usual way, such as CLEAR 1Ø0, the action is to reserve bytes at the highest memory addresses in RAM for string storage. CLEAR 1Ø0, for example, reserves space for 1Ø0 bytes of string characters. We can combine this with another type of clear operation, however. If, for example, you type CLEAR 1Ø0,327Ø0 then, in addition to leaving room for 1Ø0 bytes of string characters, you have ensured that the Dragon cannot use any address higher than 327Ø0 for its own purposes. This leaves addresses from 327Ø1 to 32767 for your machine code programs, or for storing any bytes that you want to be secure. 32767 is the highest address of RAM on the 32K Dragon. Note that the CLEAR instruction has used denary numbers, but we could have used hex provided we remembered to use &H preceding each number. In hex, 32767 is $7FFF.

The other problem is how to place the starting address for your program into the program counter of the 6809. Fortunately, the designers of the Dragon have been kind to you. There is a BASIC command EXEC which will do this for you. EXEC has to be followed by a number, and this number will be placed into the PC. It will, therefore, be used as the address of the first byte of your program. Incidentally, I've taken this as meaning 'starting byte'. It's possible to write programs in which the first few bytes are data, so that the program starts at, say, the tenth byte. This creates no problems; you simply use the address of the starting byte as the number for EXEC.

Lastly, for the moment at least, you have to ensure that your machine code program will stop in an orderly way. Nothing that we have done so far will indicate to the 6809 of the Dragon where your program ends. As a result, the 6809 could continue to read bytes after the end of your program, until it encounters some byte which causes a 'crash'. This might, for example, be a byte which causes an endless loop. Some programmers doubt if there are any bytes which do not cause an endless loop in these circumstances! To return correctly to the operating system of the Dragon, you need to end each machine code program with a 'return from subroutine' instruction, whose mnemonic is RTS and whose code is $39.

There's another headache that we don't have to worry about at the

moment. When you run a machine code program along with a BASIC program in your Dragon, you are using the same 6809 microprocessor for both jobs. It can't cope with both at the same time, so it runs one, then the other. If you make use of the 6809 registers in your machine code program, as you are bound to do, then you have to be quite certain that you are not destroying information that the BASIC program needs. Suppose, for example, that the registers of the 6809 contained the address of a reserved word in the ROM at the instant when your machine code program started. It will, therefore, need this address in these registers when your machine code program ends. This is taken care of automatically when a machine code program is called into action by using the EXEC command. The contents of the registers of the 6809 are placed into a part of the RAM memory which is called the *stack*. This, incidentally, is another good reason for being careful as to where you place your machine code in the memory. If you wipe out the stack, the Dragon will quite certainly not like it! When the RTS instruction is encountered at the end of your machine code, the bytes that have been stored in the stack are replaced into their registers, and normal action resumes. If you call a machine code program into action by any other method, not using EXEC, you will have to attend to this salvage operation for yourself as part of your machine code program. This involves using the PUSH (PSH) and PULL (PL) commands – but more of that later.

Practical programs at last

With all of these preliminaries out of the way, we can at last start on some programs which are very simple, but which are intended to get you familiar with the way in which programs are placed into the memory of the Dragon. You will also get some experience in the use of assembly language and machine code, and with how a machine code program can be run.

We'll start with the simplest possible example – a program which just places a byte into the memory. In assembly language, it reads:

```
ORG 327Ø1;   start placing bytes here
LDA #$55;    place hex 55 in accumulator
STA $7FEE;   store them at 7FEE
RTS;   go back to BASIC
```

The first line contains a mnemonic, ORG, which you haven't seen

before. It isn't part of the instructions of the 6809, but it is an instruction to the assembler, which in this case is you! ORG is short for origin, and it's a reminder that this is the first address that will be used for your program. We've chosen to use an address which leaves space for longer programs than we shall be writing in the course of this book, and we could have chosen a higher number. It will do as well as any other, however, and it leaves plenty of room for longer programs. When you program using an assembler, this line can be typed and the assembler will then automatically place the bytes of the program in the memory starting at this address. As it is, with assembly being done 'by hand', it simply acts as a reminder of what addresses to use. Note the comments which follow the semicolons. The semicolon in assembly language is used in the same way as a REM in BASIC. Whatever follows the semicolon is just a comment which the assembler ignores, but which the programmer may find useful.

Now we need to look at what the program is doing. The first real instruction is to load the number $55 into the 'A' accumulator. This uses immediate addressing, so the number $55 will have to be placed immediately following the instruction. The hashmark, '#' is used in assembly language to indicate that immediate loading is to be used. The next line commands the byte in the accumulator (now $55) to be stored at address $7FEE. In denary, this is 32750. It's an address well above the ones that we shall use for the program. Obviously, we wouldn't want to use an address which was also going to be used by the program. This instruction uses extended addressing. There are more elegant methods, but not for beginners! Finally, the program ends with the RTS instruction, essential for ensuring that Dragon life continues normally after our program ends.

The next step in programming is to write down the codes. Each code has to be looked up, taking care to select the correct code for the addressing method. The code for LDA immediate is $86, so that is the first byte of the program which will be stored at address (denary) 32701. We can start a table of address and data numbers with this entry:

```
32701    $86
```

and then move on. The byte that we want to load is $55, and this has to be put into the next memory address, because this is how immediate addressing works. The table now looks like this:

```
32701    $86
```

32702 $55

The next byte we need is the instruction byte for STA, with extended addressing. This byte is $B7, and it has to be followed by the two bytes of the address at which we want the bytes stored. The address 32750 translates into hex as $7FEE, so we can use the bytes $7F and $EE following the STA instruction. The last code has to be the RTS code of $39, so that the table now looks as in Fig. 5.4. It uses addresses 32701 to 32706, six bytes in all, and will place a byte into 32750, using denary numbers. Now we have to put it into memory and make it work!

Denary Address	Code (Hex)
32701	86
32702	55
32703	B7
32704	7F
32705	EE
32706	39

Fig. 5.4. The coded program, using denary addresses and hex bytes of data

This requires a BASIC program which will clear the memory, and poke the bytes in one by one. The program is shown in Fig. 5.5. By using CLEAR50,32700, we ensure that all memory addresses above 32700 are left unused by Dragon. That means addresses 32701 to 32767 in denary. We declare the variable A as 32700, so that we can make use of this in the POKE commands. Lines 20 to 40 then poke data numbers into addresses that start at 32701. Why 32701? Well, we have used POKEA+N, and with A=32700 and N=1, the first address just has to be 32701. The POKE part of the line uses VAL("$H"+D$) to find the number value of the hex codes so that they are poked in denary. We could, of course, have written the whole program in denary, but it causes complications when we come to write address numbers in two-byte form. Since you have to make use of hex when you graduate to the DASM assembler, or any other

```
10 CLEAR50,32700:A=32700
20 FOR N=1TO6
30 READ D$:POKE A+N,VAL("&H"+D$)
40 NEXT
50 EXEC 32701
100 DATA 86,55,B7,7F,EE,39
```

Fig. 5.5. The BASIC program which pokes the bytes into place. Note how the value of the hex codes is poked by using VAL("&H"+D$)

assembler, it's as well to start getting familiar with the principles now.

The last program line, line 5Ø, contains EXEC327Ø I. This is the BASIC instruction which will cause your machine code program to run, with the start address specified. Line 1ØØ then contains the six bytes of data that we have worked out. When you RUN this, there's no obvious effect. That's because you can't see what's in address 3275Ø. If you use:

```
?PEEK(3275Ø)
```

then you should find the value of 85, which is the denary version of $55, the number that the program put there. Now try this: type POKE3275Ø,255, and then delete line 5Ø of your program. This is the EXEC line. RUN the program again, and use ?PEEK(3275Ø) to find what's there. It should be 255. Now type EXEC327Ø I and press ENTER. Using ?PEEK(3275Ø) should now give you 85 again. This is because poking the bytes of the program into memory won't make the program run, only EXEC does this. You can therefore poke values into memory early in a BASIC program, and then make use of them later with an EXEC wherever you like.

Now this isn't an ambitious piece of work, it does no more than POKE3275Ø,85 would do in BASIC, but it's a start. The main thing at this point is to get used to the way in which machine code operates, and how you place it into memory and run it. Another point, incidentally, is that the machine code is safe in memory. If you type NEW (ENTER), the BASIC program will be cleared out, but your machine code remains. If you POKE3275Ø,255 now, and test with ?PEEK(3275Ø), you will find that this address can still be changed by using EXEC327Ø I. These bytes will remain there until you make an effort to change them, or you switch off. You can preserve the machine code program on tape if you like, and this is a technique that we'll look at later. One step at a time, if you please! Another thing we'll leave unsaid in this book is the alternative method of calling up a program, using the USR command.

Now let's try something a lot more ambitious in terms of our use

```
LDX   #$7FEE
LDA   Ø,X
ASLA
STA   1,X
RTS
```

Fig. 5.6. The assembly language program for 'multiply by two'.

of machine code – though the example is simple enough. Figure 5.6 shows the assembly language version of the program. What we are going to do is to load a byte into the accumulator, shift it one place left, and then put it into memory at an address one step higher than the address from which we took it. This looks like an open-and-shut case for indexed addressing, so we shall start by placing an address into the X register. This is the LDX #$7FEE step. As before, the '#' means immediate addressing. The next line, LDA Ø,X means that the accumulator is to be loaded from the address in the X register, with Ø added to this number. This makes the load from $7FEE (32750 denary). The third step is ASLA, arithmetic left shift, so that the bits of the byte are shifted left. Fourthly, we store this at address $7FEF by using STA 1,X. This time, 1 is added to the number (which is $7FEE) in the X register, so that the byte is stored at address $7FEF. We end, as always, with the RTS instruction.

Now we can put this into code form. It's not quite so easy as before, because of the use of indexing. The LDX instruction needs the immediate loading code of $8E, and this has to be followed by the two bytes of the address, $7F and $EE, in that order. The LDA with indexed addressing is coded as $A6, but that isn't the end of it. When you use an indexed instruction of this sort, the instruction byte has to be followed by another byte. This, called the *post-byte* is needed to specify essential information, such as which register to use for indexing. In this example, we are using numbers like Ø and 1, and we want to use the X register. Now since the numbers that we want to use as offsets are small, and will code in less than five bits (a range of −16 to +15), then the first bit of the post-byte can be zero. Since we are using the X register as the index (not Y, U or S), then the next two bits are also Ø. The last five bits of the byte are then the displacement written in binary. For a zero displacement, this is ØØØØØ, so that the whole byte in binary is ØØ ØØØØØØ, or $ØØ in hex. Easy enough! The ASLA byte is 48, and then we get to the STA 1,X. The STA indexed part of it is $A7, and then we need another of these post-bytes. The only difference between this one and the previous one is that we want a displacement of 1 rather than of Ø, so the number is $Ø1 rather than $ØØ. We don't have to go to the binary version to work this one out! Finally, 39 is the RTS command.

Now we have to code this in BASIC. If we choose a small number to place into $7FEE, the effect of the left shift will be to double the number, so we can use this to obtain a bit of arithmetic wizardry. The BASIC program is shown in Fig. 5.7. We start, as usual, by clearing memory space. You needn't worry if you have had another

```
10 CLEAR50,32700:A=32700
20 FOR N=1TO9
30 READ D$:POKEA+N,VAL("&H"+D$)
40 NEXT:POKE&H7FEE,20
50 EXEC32701
60 PRINT"2 TIMES ";PEEK(&H7FEE);" IS ";P
EEK(&H7FEF)
100 DATA 8E,7F,EE,A6,00,48,A7,01,39
```

Fig. 5.7. The BASIC program which pokes the bytes into place and then makes use of the machine code program.

program in this part of the memory before. The new program will replace it completely and, provided that your program ends correctly with the RTS instruction byte, the old program bytes cannot interfere with the new ones. The values are poked into place in the usual way in lines 2Ø to 4Ø. In line 4Ø, however, we place a number, 2Ø denary, into the address $7FEE. Now this is the address which will be used by the program, and the byte which is 2Ø in binary form, ØØØ1Ø1ØØ, will be placed in this address. In line 5Ø, EXEC327Ø1 will carry out the machine code program, which should left-shift this byte, making it ØØ1Ø1ØØØ. In denary, this is 4Ø, twice 2Ø. Line 6Ø prints this result, and line 1ØØ contains the data bytes.

It's simple enough but, if you knew nothing about machine code, you would wonder how on earth the number became multiplied by two. Once again, the program does nothing that could not be done more easily and as quickly by using BASIC. The important thing, from our point of view, is that you have now used indexed addressing and a shift instruction, as well as getting more experience in putting a machine code program into your Dragon by the hardest method of all. If, incidentally, you have made any mistakes, particularly with DATA, then it's likely that the Dragon will go into a trance and refuse to do anything. When you have typed in a BASIC program like this which pokes bytes into the memory, always record the BASIC program before you RUN it. This way, if the effect of an incorrect byte is to zonk out half the RAM, you can switch off, then on again, and reload your program. If you didn't record it, then you'll have to type it all over again. That's hard work. I know – I've just done it, having put 37 instead of 39 for the RTS byte. Yes, I know I should have recorded it. Yes, I am! Make sure you do better.

Chapter Six

Taking a Bigger Byte

The simple programs that we looked at in Chapter 5 don't do much, though they are useful as practice in the way that machine code programs are written. Practising assembly language writing and its conversion into machine code is essential at this stage, because you can more easily find if you are making a mistake when the programs are so simple. It's not so easy to pick up a mistake in a long machine code program, particularly when you are still struggling to learn the language!

Most beginners' difficulties arise, oddly enough, because machine code is so simple, rather than because it is difficult. Because machine code is so simple, you need a large number of instruction steps to achieve anything useful, and when a program contains a large number of instruction steps, it's more difficult to plan. The most difficult part of that planning is breaking down what you want to do into a set of steps that can be tackled by assembly language instructions. For this part of the planning, flowcharts are by far the most useful method of finding your way around. I never think that flowcharts are ideally suited for planning BASIC programs, but they really come into their own for planning machine code.

Flowcharts

Flowcharts are to programs as block diagrams are to hardware – they show what is to be done (or attempted) without going into any more detail than is needed. A flowchart consists of a set of shapes, with each shape being the symbol for some type of action. Figure 6.1 shows some of the most important flowchart shapes for our purposes (taken from the British Standard set of flowchart shapes). These are the terminator (start or end), the input or output, the process (or action) and the decision steps. Inside the shapes, we can

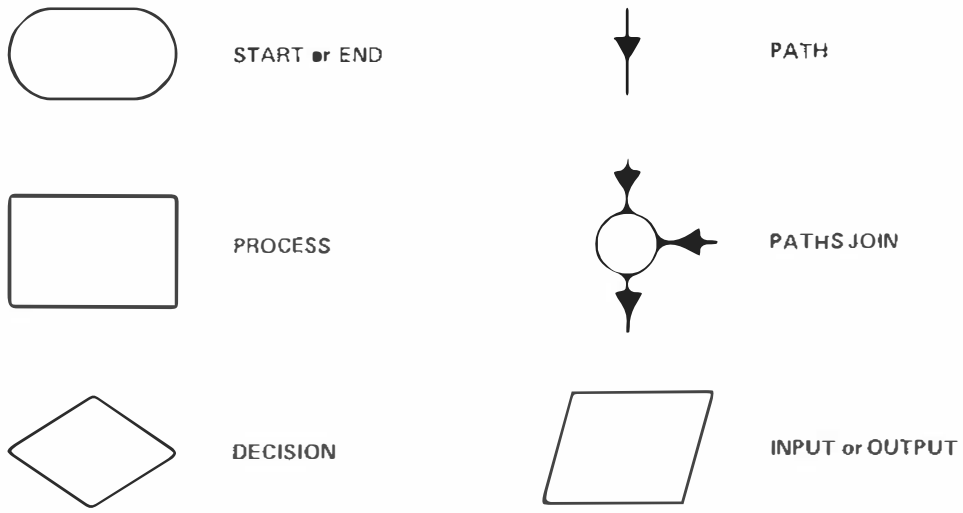


Fig. 6.1. The main flowchart shapes.

write brief notes of the action that we want, but once again without details.

An example is always the best way of showing how a flowchart is used. Suppose that you want a machine code program that takes the ASCII code for a key that has been pressed, and prints the character corresponding to that key. A flowchart for this action is shown in Fig. 6.2. The first terminator is 'START', because every program or piece of program has to start somewhere. The arrowed line shows that this leads to the first 'action' block, which is labelled 'get character in A'. This describes what we want to do – get the code number for a character in the accumulator. We don't know how we're going to do this at present – that comes later. After getting the character, the arrow points to the next action, storing the byte in

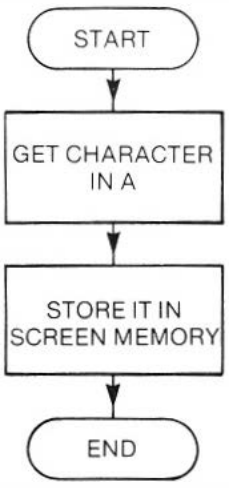


Fig. 6.2. A flowchart for the 'print-a-character' program.

screen memory. That's how we carry out the 'print' part of the action, and it's something that we've looked at earlier. The END terminator then reminds us that this is the end of this piece of program, it's not an endless loop.

This is a very simple flowchart, but it is enough to illustrate what I mean. Note that the descriptions are fairly general ones – you don't ever put assembly language instructions inside the boxes of your flowchart. Strictly speaking, I should not have referred to the accumulator A in the 'get character' box, but my excuse is that I need to be reminded of where the code is to be stored. A flowchart should be written so that it will show anyone who looks at it what is going on. It should never be something that only the designer of the program can understand and use, and just confuses anyone else. A good flowchart, in fact, is one that could be used by any programmer to write a program in any variety of machine code – or in any other computer 'language', such as BASIC, FORTH, PASCAL and so on. A lot of flowcharts, alas, are constructed after the program has been written (usually by lots of trial and error) in the hope that they will make the action clearer. They don't, and you wouldn't do that, would you?

Once you have a flowchart, you can check that it will do what you want by going over it very carefully. In the example, the actions of 'get character' and 'store in screen memory' are going to be done using machine code, so we'll concentrate on them. Getting the ASCII code for a character looks tricky at first. A lot of computers, however, put the ASCII code for the last character that was used into an address in memory. This is where a good knowledge of the way that the Dragon uses its memory comes in handy! As it happens, if you have my book *The Dragon32 And How to Make The Most Of It*, you will find some of these addresses in it, and Appendix F also shows some, a longer list in this case. One is of particular interest, the address $ØØ87. This is the address at which the Dragon stores a code number for the last key that was pressed. If we load the accumulator from this address we may, as the advertisements say, learn something to our advantage. The first step, then, looks like loading the accumulator, using extended addressing, so that we can make use of address $ØØ87. Wait, though – why should we use extended addressing for an address that starts with ØØ? Any address that starts with ØØ is a zero-page address, so we could use the shorter and quicker method of zero-page addressing. Now we're beginning to think like machine code programmers! In doing so, however, we may have dug a trap for ourselves. The direct page addressing of the

Dragon uses the number stored in the DP register as the high byte of any address that uses DP addressing. If the DP register of the Dragon happens to be set to ØØ, we can use this type of addressing. If it isn't, we'll have to use extended addressing. Right now, we can't tell what the Dragon designer did – so we'll have to try it and see!

The second step, of storing the byte in the screen memory, is straightforward. The memory that we shall use is the 'text memory' which is in the range of $Ø4ØØ to $Ø5FF. How about a spot right in the centre of the screen, at $Ø51Ø? You want to know how I got to that number? Well, if we take the range $Ø4ØØ to $Ø5FF, that's $Ø2ØØ addresses, including the first and the last. Half of that is $Ø1ØØ, so if we add $Ø1ØØ to $Ø4ØØ, we get $Ø5ØØ, which must be the first address in the middle line on the screen. There are $2Ø characters per line (all right, 32 if you're still in denary), so $1Ø is the centre of that line. Adding these gives us the $Ø51Ø address. A 'store accumulator to $Ø51Ø' should therefore get us where we want.

Now we can design the assembly language part of the code. We can follow the path we have trod before, and start the code at address 327Ø1 (denary). This makes our assembly language code look like this:

```
ORG 327Ø1
LDA $87
STA $Ø51Ø
RTS
```

We can then put this into the form of a BASIC program which pokes the codes into memory, and then calls the machine code program. It will look as Fig. 6.3, so we can enter it and run it. Another small step for a Dragon user!

```
10 CLEAR50,32700:A=32700
20 FOR N=1 TO 6
30 READ D$:POKE A+N,VAL("&H"+D$)
40 NEXT:EXEC 32701
50 DATA 96,87,87,05,10,39
```

Fig. 6.3. The BASIC poke program for printing a character.

Well, it works, but not as we might expect. When the program runs, an inverse-video @ sign appears in the middle of the screen. The problem is – where did this come from? Unless you know about the workings of the Dragon, it's not exactly easy to explain. Light comes streaming in, however, when you take a look at the tables in Fig. 6.4. These show how the Dragon responds to code numbers that

Character	POKE		CHR$	
	Dec.	Hex	Dec.	Hex.
@	00	00	-	-
A	01	01	97	61
B	02	02	98	62
C	03	03	99	63
D	04	04	100	64
E	05	05	101	65
F	06	06	102	66
G	07	07	103	67
H	08	08	104	68
I	09	09	105	69
J	10	0A	106	6A
K	11	0B	107	6B
L	12	0C	108	6C
M	13	0D	109	6D
N	14	0E	110	6E
O	15	0F	111	6F
P	16	10	112	70
Q	17	11	113	71
R	18	12	114	72
S	19	13	115	73
T	20	14	116	74
U	21	15	117	75
V	22	16	118	76
W	23	17	119	77
X	24	18	120	78
Y	25	19	121	79
Z	26	1A	122	7A
[	27	1B	123	7B
\	28	1C	124	7C
]	29	1D	125	7D
^	30	1E	126	7E
←	31	1F	127	7F
	32	20	No	
!	33	21	CHR$	
"	34	22	equivalents	
#	35	23		
$	36	24		
%	37	25		
&	38	26		
'	39	27		
(	40	28		
)	41	29		
*	42	2A		
+	43	2B		
,	44	2C		
-	45	2D		
.	46	2E		
/	47	2F		
0	48	30		
1	49	31		
2	50	32		
3	51	33		
4	52	34		
5	53	35		
6	54	36		

Fig 6.4

Character	POKE		CHR$		
	Dec.	Hex.	Dec.	Hex.	
7	55	37			
8	56	38			
9	57	39			
:	58	3A			
;	59	3B			
<	60	3C			
=	61	3D			
>	62	3E			
?	63	3F			
@	64	40	64	40	Upper
A	65	41	65	41	case
B	66	42	66	42	
C	67	43	67	43	
D	68	44	68	44	
E	69	45	69	45	
F	70	46	70	46	
G	71	47	71	47	
H	72	48	72	48	
I	73	49	73	49	
J	74	4A	74	4A	
K	75	4B	75	4B	
L	76	4C	76	4C	
M	77	4D	77	4D	
N	78	4E	78	4E	
O	79	4F	79	4F	
P	80	50	80	50	
Q	81	51	81	51	
R	82	52	82	52	
S	83	53	83	53	
T	84	54	84	54	
U	85	55	85	55	
V	86	56	86	56	
W	87	57	87	57	
X	88	58	88	58	
Y	89	59	89	59	
Z	90	5A	90	5A	
[	91	5B	91	5B	
\	92	5C	92	5C	
]	93	5D	93	5D	
^	94	5E	94	5E	
←	95	5F	95	5F	
	96	60	32	20	
!	97	61	33	21	
"	98	62	34	22	
#	99	63	35	23	
$	100	64	36	24	
%	101	65	37	25	
&	102	66	38	26	
'	103	67	39	27	
(	104	68	40	28	
)	105	69	41	29	
*	106	6A	42	2A	
+	107	6B	43	2B	
,	108	6C	44	2C	
-	109	6D	45	2D	
.	110	6E	46	2E	
/	111	6F	47	2F	
0	112	70	48	30	
1	113	71	49	31	

Fig. 6.4. (contd.)

Character	POKE		CHR$	
	Dec.	Hex	Dec.	Hex.
2	114	72	50	32
3	115	73	51	33
4	116	74	52	34
5	117	75	53	35
6	118	76	54	36
7	119	77	55	37
8	120	78	56	38
9	121	79	57	39
:	122	7A	58	3A
;	123	7B	59	3B
<	124	7C	60	3C
=	125	7D	61	3D
>	126	7E	62	3E
?	127	7F	63	3F

Fig. 6.4. The characters of Dragon, and the POKE and CHR$ numbers which produce them.

are put into it in different ways. One column shows the result of using CHR$ in BASIC to place a number into character form. The other column shows what you get when you use POKE or its machine code equivalent, which is storing the accumulator to the screen memory. The inverse (or lower case) @ has a POKE code of Ø. When we get an inverse @ on the screen from this program, then, it must be because the number that was in the accumulator was zero. That makes sense, because we hadn't pressed any key when the program ran. This program is the equivalent of using K$=INKEY$ without putting the instruction in a loop. What we have to do is to loop around until the number is no longer zero.

Don't rush – we have to check this. Try this one-liner in BASIC:

```
1ØØ K=PEEK(&H87):IF K=Ø  THEN 1ØØ ELSE PRINT K
```

Now when we run this one the program hangs up, waiting for us, and when we press a letter key, we see the ASCII code for the letter appear on the screen. That confirms what we suspected – the address $87 will hold zero until a key is pressed, and to make it work the way we want, we need to loop around until the address no longer stores a zero.

Now in our present state of knowledge, we could do the looping in BASIC, and call the machine code program as soon as the BASIC had detected a key being pressed. Rather than waste time over this, though, let's try the complete machine code approach, even at the risk of making some mistakes on the way.

Loop back in hope

Since this is a simple program, it looks like a good opportunity to get

an introduction to looping. If you have done anything more than the most elementary BASIC programming, you will know what a loop involves. A loop exists when a piece of program can be repeated over and over again until some test succeeds. In BASIC, you can cause a loop to happen by using a line which might read, for example:

```
2ØØ IF A=Ø THEN GOTO 1ØØ
```

This contains a test (is A=Ø?), and if the test succeeds (yes, A is Ø), then the program goes back to line 1ØØ and repeats all the steps from there to line 2ØØ again. That sort of loop in BASIC corresponds very closely to how we create a loop in machine code. Instead of using line numbers, however, we are using address numbers. Instead of testing a variable called 'A', we shall test the contents of a register, which in this case can be the 'A' register.

Let's start the proper way with a flowchart. Figure 6.5 shows how

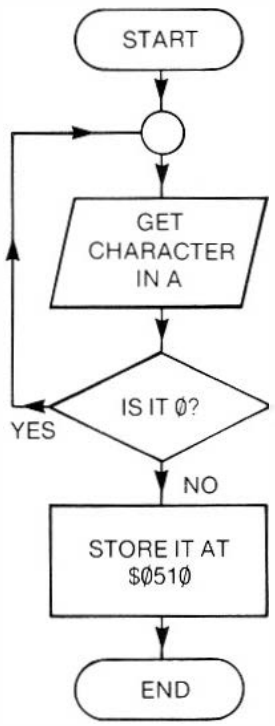


Fig. 6.5. Another flowchart – this one has a loop which rejects the zero character.

this might look. The first step is the same – get the character code in the accumulator A. The next step, however, is a 'decision' step. The decision is: 'is it Ø?' All decision steps in flowcharts must be worded so that there can only be two possible answers, yes or no. This is indicated by having two arrowed paths from the decision step. One of these is labelled 'YES'. It leads back to the first step of the

program, the step that requires the memory to be loaded to the accumulator. Why? Because if we find that we have a zero in the accumulator it means that there's no key pressed, and we have to go back and try again. The other path, the one that is labelled 'NO' leads to the next action step, storing the accumulator byte in the screen memory.

The action, then, will be that the accumulator is loaded from address $ØØ87, and the byte in the accumulator is tested to see if it is zero. If it is, we repeat the loading. If it isn't (which means that a key was pressed), then we store the byte in the screen memory. Now we have to put this into assembly language form – and that's going to introduce some new items to you.

```
LOOP: LDA $87     ; load accumulator from address
                    ØØ87
      BEQ LOOP    ; go back if the byte is Ø
      STA $Ø51Ø   ; otherwise store it at address
                    $Ø51Ø
      RTS         ; back to BASIC
```

Fig. 6.6. The assembly language program corresponding to the flowchart.

Figure 6.6 shows an assembly language program which should carry out the effect of the flowchart. There's one more step in this flowchart, and one alteration to an existing step. The new step is the 'BEQ LOOP', and the change is to the first step, which now has LOOP: stuck in front of it. This word LOOP is a *label*. It's being used here in place of an address, and it means the address at which the instruction starts. With LOOP: placed in front of the LDA $87 instruction, the word loop means the address at which the LDA instruction byte is stored. By using words in this way, we avoid having to think about address numbers until we actually write the machine code. If we use an assembler, we usually don't have to worry about address numbers at all – the assembler automatically puts in address numbers in place of label words. The same label word is also used in the next step. BEQ means 'branch if the register is equal to zero', so the effect of BEQ LOOP is that the program should go back to the address of the LDA instruction if the accumulator contains zero. It's rather like using a version of BASIC which allowed variables to be used in place of line numbers (as some do).

In assembly language, this all looks quite neat and straightforward. If we were using an assembler it would be straightforward, but when we assemble by hand, it's not so simple. The reason is that we have to

```
Destination - address you want to jump to (which has label in
              front of assembly instruction)

    Source - address you want to jump from (address of branch
             code - in assembly language; it is the instruction
             with the label name after it)

Displacement is Destination minus Source minus 2 put into hex form.

Checking:       Forward displacement      Codes        Calculation
Address

                                                       327Ø8 - 327Ø3 - 2 = Ø3

327Ø1           LDA #$Ø6                  86 Ø6        Check
327Ø3           BRA START                 2Ø Ø3 <----   displacement Ø3 - jump over
327Ø5           LDX #$7FFØ                8E 7F FØ     3 bytes to A7
327Ø8 START:    STA 1,X                   A7 Ø1

                Backward displacement                  Calculation

                                                       7FFØ - 7FF5 = -5
7FFØ    LOOP:   LDA $7FØØ                 B6 7F ØØ     - 5 - 2 = -7 = F9
7FF3            CMPA #$Ø3                 81 Ø3
7FF5            BEQ LOOP                  27 F9        Checking
                                                       Start counting at the F9
                                                       byte, and count up for each
                                                       byte to $B6. You should reach
                                                       FF when you get to $B6.
```

Fig. 6.7. The formula for finding the size of a displacement byte.

follow the BEQ instruction by a single byte which will give the address of the LDA instruction. This is PC-relative addressing, so that we have to use a signed byte that can be added to the address in the program counter to give the address of the LDA step. The formula is shown in Fig. 6.7. What you have to do is to find the address that you want to jump to, and the address of the branch command. Subtract these, then subtract 2 from the result. What you have now is the size of the 'displacement' byte that you need to follow the branch instruction. Since this number is negative, we have to convert it to the form of a signed byte, using the procedure that we looked at earlier.

If all that sounds complicated, take a look at it in practice, in Fig. 6.8. Assuming that we are going to place the first byte of the program at address 327Ø1 then the address of the BEQ instruction is at 327Ø3. 327Ø3 is the source address, where we're coming from, and 327Ø1 is

```
                Address   Code

Destination     327Ø1     96          1. Destination - source   = 327Ø1 - 327Ø3
                327Ø2     87             address       address
Source          327Ø3     27                                    = -2
                327Ø4     this is where
                          displacement   2. Subtract 2 from this to get -4
                          byte fits
                                         3. 256 - 4 = 252 (negative form in denary)

                                         4. 252 denary = $FC and this is the
                                            displacement byte
```

Fig. 6.8. An example of finding a displacement byte.

the destination address, where we're going to. Subtract source from destination numbers, and we get −2. Subtract another 2 from this, and we get −4. −4 in hex is FC, so that's the displacement byte that is placed following the BEQ instruction code.

Now don't rush and try this – because it won't work! It's not all that easy to see why it won't work because, once more, it depends on knowing how the Dragon works. If you have, in fact, figured it out, award yourself a gold star. What is happening is that our program loops round forever. It should loop until a number code is put into address $ØØ87, and a code should be put into that address by the act of pressing a key. Why doesn't it work? Because if the microprocessor of your Dragon is spending all its time looping round your program, it can't be scanning the keyboard looking for a key to be pressed! There's only one microprocessor in the Dragon, and it has to do everything. As it happens, it doesn't have to keep the screen display going – if it did, you would see the screen picture disappear when you ran this program. Instead, you get – nothing. You'll have to use the reset button at the side of Dragon to get out of the loop.

How do we get round this? What we have to do is to write a piece of program that will attend to reading the keyboard, and place that piece of program in our loop. That's possible, but it takes a lot of

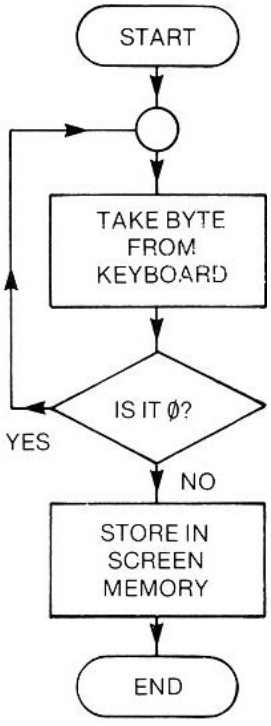


Fig. 6.9. A flowchart for a character printing program.

time, and needs a lot of knowledge of the Dragon. It also seems a trifle unnecessary, because there must be a routine in the ROM of the Dragon which will do all this for us. There is – and Appendix F lists the address for this and other useful routines. The routine which starts at address $8ØØ6 will scan the keyboard looking for a key being pressed. If no key is pressed, the number in the accumulator will be zero. If a key is pressed, the number in the accumulator will be the number code for that key. Using this routine, we don't have to make use of any memory address like $ØØ87.

The next step, then, is to see how this routine at $8ØØ6 can be used. The instruction that we need is called 'jump to subroutine', and it's abbreviated to JSR. Each subroutine in the ROM ends with the RTS code, which returns it to whatever program called it, so if we use JSR followed by the address $8ØØ6, then the subroutine will run and then return to our own program. Figure 6.9 shows a flowchart for what we are going to attempt now. We shall call up the subroutine to get the byte into the accumulator, then test it. If the byte is zero, we shall return to the 'scan keyboard' step; if not, we shall store it in screen memory. So far, so good. Figure 6.10 shows

```
LOOP:   JSR $8ØØ6   ; keyboard subroutine
        BEQ LOOP    ; back if zero
        STA $Ø51Ø   ; store to screen
        RTS         ; back to BASIC
```

Fig. 6.10. The assembly language version of the program.

the assembly language version of this flowchart, with the label word LOOP once more used to indicate the address to which the program is to return if the byte in the accumulator is zero. Figure 6.11 shows the program put into the form of a set of BASIC poke instructions. When this runs, pressing a key will cause a letter or other character to appear on the screen. Not all keys will produce a result (find out which ones don't), and some keys produce unexpected effects. Try the number keys, for example, and also try the effect of pressing the SHIFT key along with other keys. It works – and this is the machine code equivalent of the INKEY$ instruction of BASIC. We've

```
10 CLEAR50,32700:A=32700
20 FOR N=1TO9:READ D$
30 POKE A+N,VAL("&H"+D$):NEXT
40 EXEC 32701
50 DATA BD,80,06,27,FB,B7,05,10,39
```

Fig. 6.11. The BASIC poke program.

broken a lot of new ground in this short piece of program, so perhaps this is a good time to go over it all carefully and make sure that you know what it has all been about before we plunge deeper into the business.

More loops

The loop that we have tried out was a simple loop that is classified as a 'holding loop'. Its job was to keep a piece of program repeating until something happened. It's time now to take a look at another type of loop, called a 'counting loop'. The importance of this one is twofold – it's the way that we program a time delay in machine code, and it also gives me an excellent opportunity to demonstrate just how fast machine code can be.

The type of loop that you use most in BASIC is the FOR...NEXT loop. This uses a 'counter' variable to keep a score of how many times you have used the loop, and compares the value of the counter with the limit number that you have set each time the loop returns. Now the action of a FOR...NEXT loop can be simulated in BASIC without using FOR or NEXT, and the method is shown in Fig. 6.12.

```
10 C=0:ND=10
20 PRINT"ACTION ";C
30 C=C+1
40 IF C<=ND THEN 20
50 PRINT"FINISHED"
```

Fig. 6.12. A simple loop in BASIC.

The count number is C, and its limit is ND. At the end of the program, the value of C will be 11, just like the value of the counter in a FOR N=1 TO 1Ø type of loop. The next thing, then, is to take a look at the flowchart for this type of program, and that's shown in Fig. 6.13.

This method of forming a counting loop is the one that we use in machine code. We can write some assembly language that will do the same job – but, as usual, we have to give a lot more thought to how the task will be done. For one thing, we don't have variable names in machine code. We have to decide where we shall store a number, and in what register we shall carry out the task of decrementing it. The decision step is easier – we can use a BNE test this time to keep the program looping back until the content of the register that we have tested is zero. In case you're wondering how we specify which

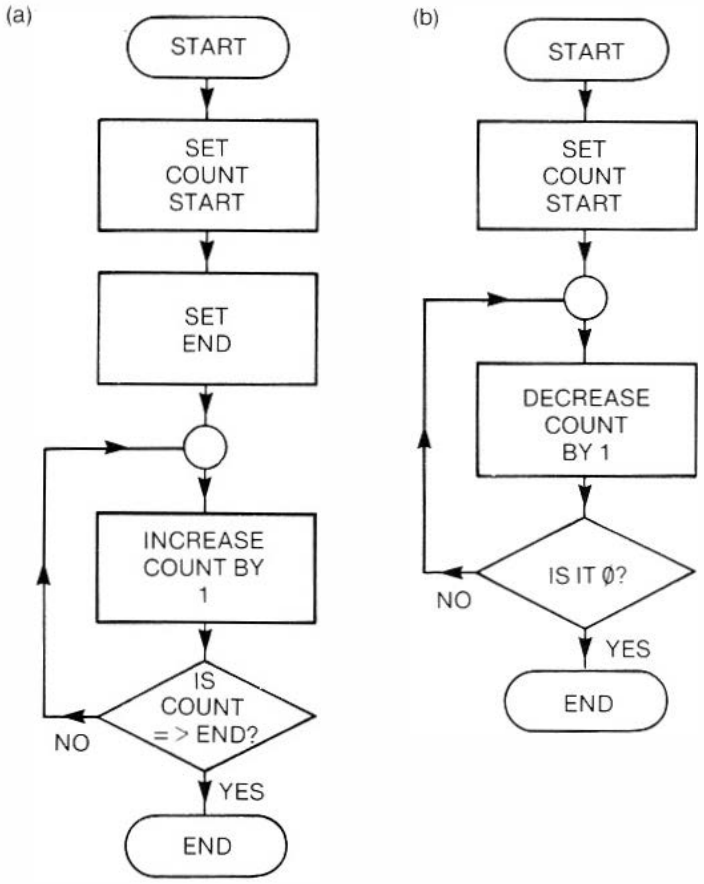


Fig. 6.13. Flowcharts for loops: (a) incrementing the count number, (b) decrementing the count number, which is simpler.

register we're testing, the answer is that it's always the one that we used just before the BNE (or any other) test.

Figure 6.14(a) shows what we end up with as an assembly language program. The accumulator is loaded with $FF, which is 255 in denary. This is the largest number that we can load into an eight-bit register. Having loaded the accumulator, we then decrement it, and mark this address as 'LOOP', the place we want to

```
        LDA #$FF     ; $FF into 'A' register
LOOP:   DECA         ; decrement it
        BNE LOOP     ; back if it has not reached Ø
        RTS          ; back to BASIC
```

(a)

return to if the register content is not zero. The test is carried out by BNE, branch if not equal to zero, because we want the program to repeat the decrementing action until the contents of the accumulator reach zero. The BASIC program which pokes the bytes into memory and then carries out the program is shown in Fig. 6.14(b). This time

```
10 CLEAR50,32700
20 FOR N=1 TO 6
30 READ D$
40 POKE&H7FBC+N,VAL("$H"+D$)
50 NEXT
60 PRINT"START":EXEC&H7FBD:PRINT"STOP"
100 DATA86,FF,4A,26,FD,39
```

(b)

Fig. 6.14. (a) A counting loop in assembly language. (b) BASIC poke program

we've used the POKE address in its hex form, and we'll do this from now on. Now when you run this one, you will not see much of a time delay between the printing of 'START' and the printing of 'STOP'. This isn't because nothing has happened, it is because the machine code countdown is so fast! If you try a BASIC version of this:

```
10 A=255:?"START"
20 A=A-1
30 IF A<>0 THEN 20
40 ?"STOP"
```

you will see that there is a noticeable pause. The difference does not reflect the comparative speeds, however, because quite a lot of time is spent in the printing actions. To see just how great the advantage of machine code can be in terms of speed, we need to work with much larger numbers. Now there are several ways of doing this, but one which we can look at right now involves two loops.

You have probably met nested loops in BASIC. The principle is that there is an inner loop and an outer loop. On each pass of the outer loop, the whole of the inner loop is carried out. This allows us to create much longer time delays, by doing one count inside another. Suppose we have, in BASIC, the lines:

```
10B=100:?"START"
20 B=B-1
30 A=255
40 A=A-1
50 IF A<>0 THEN 40
60 IF B<>0 THEN 20
70?"STOP"
```

then these would carry out a countdown of A from 255 to 0 each time the value of B was decremented. Try this one – and time it. You won't need a stop-watch – anything with a minute hand will do!

For a contrast, let's see how the same numbers could be dealt with in a machine code countdown. Figure 6.15(a) shows the assembly

```
        LDB #$64    ; B loaded with 100 (denary)
LOOP 1: LDA #$FF    ; A loaded with 255 (denary)
LOOP 2: DECA        ; A = A - 1
        BNE LOOP 2  ; decrement A if not zero
        DECB        ; now decrement B
        BNE LOOP 1  ; back for another one
(a)     RTS         ; finished
```

```
10 CLEAR50,32700
20 FOR N=1TO11
30 READ D$
40 POKE&H7FBC+N,VAL("&H"+D$)
50 NEXT
60 PRINT"START":EXEC&H7FBD:PRINT"STOP"
100 DATA C6,64,86,FF,4A,26,FD,5A,26,F8,3
(b) 9
```

Fig 6.15. (a) Assembly language for a two-loop counter, (b) the BASIC poke program.

language version. The B register is loaded with $FF, and the A register with $64 (100 denary). This second instruction is labelled 'LOOP1'. Then comes DEC A, so that the A register is decremented, and this is labelled 'LOOP2'. The BNE test then returns to this LOOP2 point until the A register has reached 0. After that, the B register is decremented, and then tested. Note that the order is not quite the same as in the BASIC version. In a machine code decrement and test action, you must have the decrement done just before the test, otherwise the register that is tested may not be the correct one. If the B register has not reached zero, the program loops back, this time to LOOP 1, to fill up the A register again and perform the 'inner loop' yet again.

When you try this – it's still almost too fast to follow! It's a good illustration of the speed advantage of machine code as compared to BASIC. If you are not quite convinced that the count has been carried out, then alter the number in the outer count from $64 to $FF (replace 64 by FF in the data line 100 of the BASIC program). This makes the delay slightly more noticeable.

Using two registers in this way is slightly clumsy, however, and it would make more sense if we could use a sixteen-bit register that we could load with $FFFF. As it happens, we can, but we can't use the DEC command with sixteen-bit registers! This might just be a good excuse to introduce some more advanced addressing as we look at a way of decrementing a sixteen-bit register.

As it happens, the A and B registers can be combined into a 'D'

register, but there is no simple way of decrementing this either. We'll use the index register, X, then. As it happens, we can load any register from the X register, using indexed addressing, and we can specify an amount added to the number in the X register. We could specify, for example, that we will load the Y register from the X register, first subtracting 1 (adding −1) from whatever is in the X register. That's one possibility – but there's another. This is to load the X register from the X register(yes, from itself!), but subtracting 1 in the process. It's just the machine code equivalent of the BASIC decrement action:

```
X=X-1
```

The command that we need for this type of action is LEA – load effective address. This is just one of the commands which can be operated by indexed addressing. What makes it different from an ordinary load is that it loads the contents of an address number rather than the number. When we specify the action:

```
LEAX -1,X
```

what we want is to take the number from the X register, add −1 to it, and then put it into the X register. It may seem a long-winded way of decrementing, but it works! There's another complication. When you look up the code for LEAX, you find that you need to follow it with another byte that will specify the other two parts of the instruction. One of these is that you want to add −1, the other is that you want to take the number from the X register so that you can add −1. All indexed addressed instructions need this second byte (called a post-byte, which sounds like the action of a dog on postmen). Figure 6.16 shows how these bytes are arrived at. There are a lot of possibilities, because there are several registers, and a lot of numbers that you could add or subtract. In this table, I've shown only the post-bytes that apply to loading from the X and Y registers with increment and decrement. The other variations come about because

		Main code	Post byte
increment X	LEAX 1,X	$3Ø	$Ø1
decrement X	LEAX -1,X	$3Ø	$1F
increment Y	LEAY 1,Y	$31	$Ø2
decrement Y	LEAY -1,Y	$31	$3F

Fig 6.16. How double-byte registers can be incremented and decremented by means of the LEA instruction.

you can choose numbers of different sizes (zero, five-bit, eight-bit or sixteen-bit) to add to the number taken from the register.

```
(a)         LDX #$FFFF  ; maximum numbers into X
      LOOP: LEAX -1,X   ; decrement X
            BNE LOOP    ; back if not zero
            RTS         ; finished
```

```
(b)  10 CLEAR50,32700
     20 FOR N=1TO8
     30 READ D$
     40 POKE&H7FBC+N,VAL("&H"+D$)
     50 NEXT
     60 PRINT"START":EXEC&H7FBD:PRINT"STOP"
     100 DATA8E,FF,FF,30,1F,26,FC,39
```

Fig. 6.17. (a) Assembly language for a countdown from $FFFF, (b) the BASIC program to poke the bytes into memory

Let's get back to the example. In the assembly language version, Fig. 6.17(a), the first step is to load the X register with the maximum size of number, $FFFF (65535 in denary). The loop starts with the indexed load, LEAX −1,X, and the usual BNE test checks to see if the result is zero. If it's not, then the program returns to the indexed load step. This continues until the X register is decremented to zero. Once again, the time delay is noticeable – but try for yourself the BASIC equivalent, which would be:

```
10 X=65536:?"START"
20X=X-1
30 IF X<>0 THEN 20
40?"STOP"
```

The result of running this should convince you of the speed advantages of machine code!

Chapter Seven

Ins and Outs and Roundabouts

Video loops

Of all the loop programs that we can make use of in machine code, loops that involve the video display addresses are among the most useful. We have seen earlier that we can make things happen on the video display by making use of POKE commands from BASIC. The techniques that you used for POKE displays on the screen can also

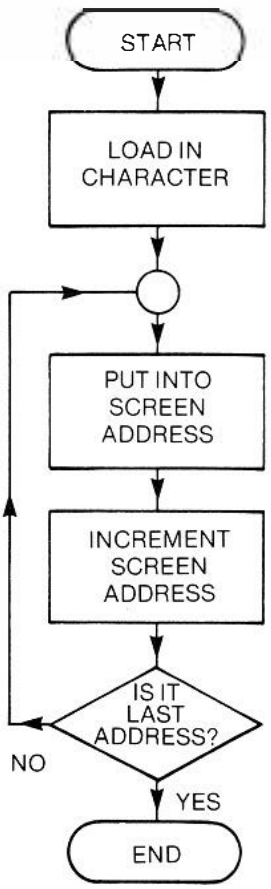


Fig. 7.1. A flowchart for filling the screen with a character.

be used for machine code, and the main differences are that machine code is faster and involves more work on your part!

Take a look, for starters, at Fig. 7.1. This shows the flowchart for a program that will fill the text screen with one character. The idea is that we load a register (the accumulator is usually the best bet) with a code number for a character, and we store this at the first screen address, which is $4Ø0. We then increment this address, compare it with the last screen address (it should be $Ø5FF, but we've gone one beyond to $Ø6Ø0 for reasons we'll explain later) and jump back if we haven't got to the last address. The jump back is to the store command, so the character gets stored at the next screen address, and so on. When we come to put this into assembly language, we'll use an indexed store command, with the automatic incrementing feature. This is why we need to use $Ø6Ø0 rather than $Ø5FF for the last screen address. If we compare the address after the increment action, storing the last character will take place when the address is $Ø5FF. This will then increment to $Ø6Ø0, and that's when we carry out the compare operation. Watch for points like this, because they

```
        LDA #$2A      ; $2A into accumulator
        LDX #$Ø4Ø0    ; first screen address
LOOP:   STA Ø,X+      ; store 2A at X address,
                        then increment X
        CMPX #$Ø6Ø0   ; reached end yet?
        BNE LOOP      ; back if not
(a)     RTS           ; all done
```

```
10 CLEAR50,32700
20 FOR N=1 TO 13:READD$
30 POKE &H7FBC+N,VAL("&H"+D$)
40 NEXT:EXEC &H7FBD
100 DATA 86,2A,8E,04,00,A7,80,8C,06,00,2
(b) 6,F9,39
```

Fig. 7.2. (a) The assembly language program, (b) BASIC poke program.

can often account for a puzzling 'spare piece' in a pattern.

Let's get down to the assembly language version, which is shown, along with the BASIC program, in Fig. 7.2. The first step is straightforward – load the accumulator with $2A. This is the ASCII code number that will produce a white-on-black star – obviously, you could try any other code number that you liked to use. We then

load the X register with the start of the screen address, which is $4ØØ. The X register is chosen for this because it's ideal for indexed storing. The next item is the loop. We start the loop by storing the byte in the accumulator to the address in the X register, and then incrementing the X register. In assembly language, it's written as:

```
STA Ø,X+
```

with the + following the X to remind you that the incrementing action is carried out after the store operation is completed. The next step is to compare this (incremented) address with $Ø6ØØ, the last number we expect to find. If the numbers are not equal (BNE), then the program loops, to store the character at another address. When the last increment action has made the address number in the X-register equal to $Ø6ØØ, then the program breaks out of the loop, and returns to BASIC.

Transforming this by hand into machine code is reasonably straightforward, except for the usual 'post-byte'. This, you remember, is the byte that we have to put following the indexed instruction. It signals to the 6809 which index register is being used, what sort of increment we want, and so on. In this case, the post-byte is $8Ø. If you used $81, you would increment the X register twice each time, so that the character was placed only in each second address. Try it later, and look at the effect.

Once the machine code bytes have been written down (check the displacement byte that follows the BNE), then the BASIC program that pokes the bytes into memory can be written down. It shouldn't take long – apart from the value of N and the DATA line, it's almost the same for each program so far. When this runs, there is a short delay while the slow BASIC pokes the numbers into the memory, and then the machine code does its stuff in its usual lightning way.

Take a bigger cast-list...

We can modify the program of Fig. 7.2 in an interesting way. Suppose we started with the accumulator containing zero, and we incremented the accumulator on each pass through the loop. This would mean that we would produce on the screen each character for the numbers Ø to 255 (denary). What would happen then? Well, since the accumulator can hold only eight bits, and 255 (denary) is the largest number it can hold, it simply goes back to Ø again next time it is incremented. Figure 7.3 shows the flowchart for this action.

and Fig. 7.4 shows the assembly language and the BASIC poke program. There's nothing here that should cause you any head-scratching, because the only addition is the INCA in the loop. Try it, and you'll see the full set of text-mode characters appear – in colour if you're using a colour TV for display. This is a good example of how a simple program can be extended so as to do much more than the original version. It's an important point, because a lot of machine code programming is of this type. If you keep a note of all the assembly language programs that you have ever used, along with what they did, then you'll find that this 'library' is a very precious asset. Very often, you'll find that any new program that you want to write can be done by modifying or combining (or both) old routines

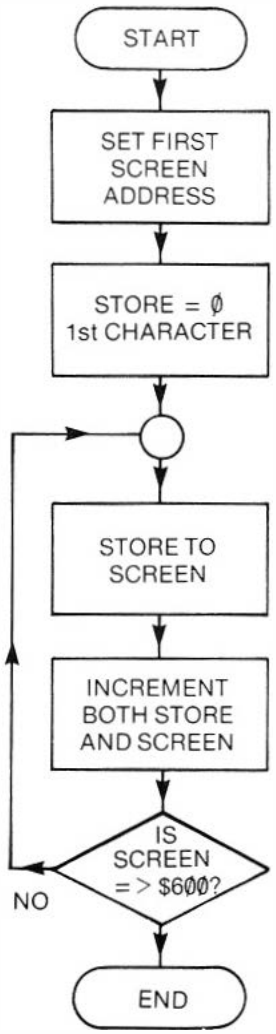


Fig. 7.3. A flowchart for printing the entire character set.

that you are familiar with. Another big advantage of this is that an old routine is a trustworthy routine – a split-new one needs a lot of testing before you can rely on it.

Save it!

At this point, when our programs are getting slightly longer, and doing more interesting things, it's time to look at the topic of saving machine code program on tape. Now you aren't absolutely obliged to do this. Our machine code programs have all been, so far, in the form of a BASIC program that poked numbers into the memory. You can, obviously, just save this BASIC program, and it will create the machine code for you whenever you want. There are times, however, when you need as much of the memory as possible, and another piece of BASIC program is as welcome as an elephant in a space capsule. You may also want to put on cassette a program that is a mixture of BASIC and machine code, and to make it difficult for anyone to copy it. Either way, you'll want to make a direct recording of the bytes that are stored in the memory. The ordinary BASIC commands of the Dragon can cope with this and, as we'll see later, we can also cause a program to be recorded or replayed by machine code commands.

We'll stick to comparatively straightforward items for the moment, however. To save a machine code program, you have to have the bytes of the machine code in the memory. You also have to know the address of the first byte of the code, and the address of the last byte of the code. Some machine code programs do not start with the first byte that is stored (they might store data first, for example), so you also need to know the starting address. This is also called the 'execution address'. Finally, you need a filename for the program. If all of this looks like a lot of bother, lets look at an example which will show that it's not so awkward as you might think.

Suppose we take the program of Fig. 7.4, which writes all the possible characters to the screen, twice. The program has its first byte, and its start at address $7FBD, and if you count in hex to the last byte, you'll find that it's at $7FCA. Alternatively, if your hex arithmetic is coming along reasonably well, you can figure that there are 13 bytes, which is $ØD, and $ØD added to $7FBD is $7FCA. If the worst comes to the worst, you could convert all the numbers to denary, add, and then convert back, or just use the denary numbers.

```
        LDX #$0400  ; start of screen into X
        LDA #$00    ; zero to accumulator
LOOP:   STA 0,X+    ; accumulator byte to screen,
                      and increment
        INCA        ; next character
        CMPX  $0600 ; end of screen?
        BNE LOOP    ; back if not
(a)     RTS         ; finished
```

```
10 CLEAR50,32700
20 FOR N=1 TO 14:READD$
30 POKE &H7FBC+N,VAL("&H"+D$)
40 NEXT:EXEC &H7FBD
100 DATA 8E,04,00,86,00,A7,80,4C,8C,06,0
(b) 0,26,F8,39
```

Fig. 7.4. (a) The assembly language version, (b) the BASIC listing for the flowchart of Fig. 7.3.

You don't have to make a meal of it, because the Dragon will do the conversions for you, and the addition as well!

To save the program, then, you use the command CSAVEM (M for machine code), and follow it with the filename (in quotes), the address of the first byte, the address of the last byte and the execution address. You must place commas to separate these items. For this example, you would need:

CSAVEM"SCREEN",&H7FBD,&H7FCA,&H7FBD

You must then press the PLAY and RECORD keys of the cassette recorder, and then press the ENTER key of the Dragon. Recording is, as you might expect, very rapid. When it's done, you can rewind the tape. Note, incidentally, that you must have the bytes in the memory before you can do this recording step!

Now comes the crunch. You have to be sure that you actually have the program on tape, and that you can load it back. This is not so easy to test, because if you just type NEW and press ENTER, all you do is to wipe the BASIC program. If you switch off, you'll certainly clear the memory, but you'll have to start again by clearing memory space. To test what you have here, the easiest way is to clear the memory that you have used, and reload the program. Type the BASIC line:

FORN=&H7FBD TO &H7FFF:POKE N,0:NEXT

and press ENTER. This will clear the memory from &H7FBD on. To prove it, type EXEC &H7FBD and press ENTER. This should cause the computer to hang up, and you'll have to rescue it by pressing the RESET button at the left-hand side. You can then reload the program. Type CLOADM"SCREEN", press the ENTER key, and press the PLAY key of the recorder. You should see the usual tape messages (S for search, then F for 'found') appearing, and when the tape stops, your program is loaded. You can now test it by typing EXEC&H7FBD, then press ENTER. If you see the screen fill with characters as it did before, all is well. You can also use EXEC by itself, because the address for starting the program (the execution address) was recorded on the tape, and it gets transferred to the EXEC routine automatically. There's a variation on the CLOADM command which allows you to load the program into a different set of memory addresses, but that's not something we want to get involved in at the moment.

It's important to note at this point that CSAVEM and CLOADM are not just commands which can be used for machine code. They are commands which save any set of bytes that happens to be stored in the memory. That includes the ASCII codes for letters, if you are interested in word processing programs, or the codes in the screen memory. You can, for example, save a good-looking screen display on tape, and replay it again to give an instant picture! All that you need to specify in the CSAVEM command, after the filename, is the start of the screen memory and the end of the screen memory. You must put in an 'execution address', and you can make this the start of screen memory. This can then be recorded in the usual way. When it's replayed, the characters that were recorded will re-appear on the screen. It's a great way of replaying a long set of instructions, or of reproducing a pattern that takes a lot of BASIC instructions to create.

Saving a screen is particularly useful when you are using the high resolution screens, because if your programs use a lot of graphics that are animated by 'page-flicking', then all of the pages of memory that you use can be saved in one piece. Since these are not displayed on the screen until you use the SCREEN instruction in BASIC, anyone using the program can't see what is being loaded in. This can be an excellent way of starting a game, or placing a logo on the screen that identifies you as the author, and so on.

Take a message ...

After that brief interlude on the subject of saving and reloading machine code, let's get back to the programs. We left Fig. 7.4, you remember, writing characters on the screen. It's time now that we looked at ways of puttingsomething more interesting on the screen, and letters look like a reasonably simple start to this type of programming. What do we have to do? Well, to start with, we need to store some ASCII codes for letters somewhere in the memory; we can't just use a string variable as we would in BASIC. We will have to know the address at which the first of the letters is stored, and how many letters are stored starting at this address. After that, we should be able to work a loop which takes a byte from the 'text space' (where the letter codes are stored) and put it into the screen space (the screen addresses). We've already used the main type of instruction that we need for this sort of thing – the auto-incremented load or save. To work, then!

We start, as always, with a flowchart. It's not so easy this time, because we need a different way of ending the loop. We could count the number of letters that we want to place on the screen, but I want to look at a different technique this time – using a *terminator*. You are probably familiar with this idea used in BASIC programs. A 'terminator' is a byte which the program can recognise as a special character, one which is not, for example, part of a message. A convenient terminator for a lot of purposes is Ø, so we'll try that. The difficulty arises because we don't want this terminator printed on the screen. Because of the way that the Dragon uses its code numbers, the number Ø actually would give us a printed character – the inverse @ that we saw earlier. We don't want this to appear, so we must test the accumulator between loading the byte from memory and placing it into the screen memory. That, as you'll see, makes the loops more complicated.

The flowchart that we need is shown in Fig. 7.5. What we have to do is to store two addresses. One of these will be familiar, it will be part of the screen memory. This has to be the address of the first byte that we will want to put on the screen. The other address has to be a store address, the start of a string of bytes that will be used to store ASCII codes, and which will not be used for anything else. We can clear things up for ourselves by giving these two addresses label names. I've chosen SCRN for the screen memory, and TXT for where we're storing the codes. What we do, then, having allocated these addresses, is to load a code from TXT, and increment the TXT

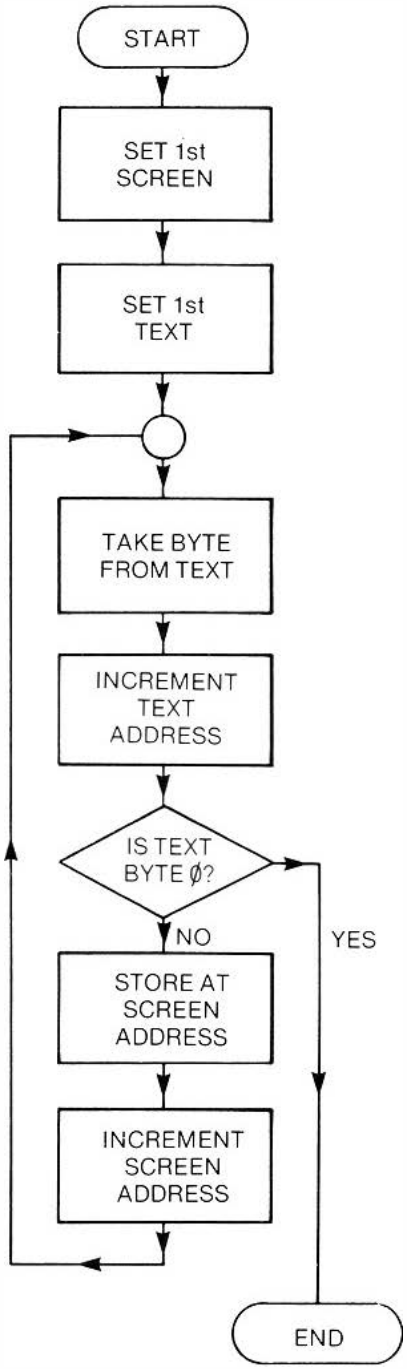


Fig. 7.5. A flowchart for printing a message on the screen

address. We then test the character, to see if it's our terminator of Ø. If it is, we want to leave the program at once. If it's not, then we store this byte at SCRN, increment the SCRN address, and go back for another character. Now this gives us a flowchart which has two jumps. One of these is the 'go back' part, the other is the part that

```
        LDX  # $Ø464   ; position on screen
        LDY  # $7FEØ   ; address of text
LOOP:   LDA   Ø,Y+     ; get character, increment
                         text
        CMPA  #$ØØ     ; is it Ø?
        BEQ   OUT      ; out if it is
        STA   Ø,X+     ; store if not zero,
                         increment screen
        BRA   LOOP     ; back for another
OUT:    RTS            ; finished
```

(a)

```
10 CLEAR50,&H7F00:A$="DRAGON MAGIC!"
20 FORN=0TO12
25 C=ASC(MID$(A$,N+1,1))
26 IF C<64THEN C=C+64
30 POKE&H7FE0+N,C:NEXT
40 POKE&H7FE0+N,0
41 REM MESSAGE
50 FORN=1TO18:READ D$
60 POKE&H7F00+N,VAL("&H"+D$):NEXT
61 REM MACHINE CODE
70 CLS:EXEC&H7F01
100 DATA8E,04,64,10,8E,7F,E0,A6,A0,81,00
,27,04,A7,80,20,F6,39
```

(b)

Fig. 7.6. (a) The assembly language for the message routine. (b) the BASIC listing

goes to theend. What is this going to look like in assembly language?

The answer appears in Fig. 7.6. It follows the flowchart pretty exactly, and the parts we particularly need to look at are the BEQ and the BRA steps. At the BEQ step, the accumulator has been loaded from the TXT piece of memory, whose starting address is held in the Y register. BEQ means 'branch if equal to zero', and the displacement that follows this instruction byte will take the program to the RTS instruction, skipping over the steps between the BEQ and the RTS. If the byte is not zero, however, it is stored at the SCRN address, using the X register with automatic increment. We then have to get back for another character. Since this needs a jump, and one that must always be made at this point, we use the BRA (branch always) instruction. Take that smirk off your face, Jones minor.

That explained, we can convert into the form of a BASIC program, and try it out. We'll place the bytes into memory in a fairly simple way, by poking them into memory from a string. This is done

in lines 2Ø to 4Ø. Lines 5Ø to 7Ø then deal with the machine code, and run it. When it runs, you see the message – but not as you might expect it! The problem lies once again, with the way that Dragon uses its codes. The code for a space, in ASCII, is 32 (denary), and the code for the exclamation mark is 33(denary). Now when you use 32 or 33 in a CHR$ instruction in BASIC, you get the correct characters. When characters are poked into the screen memory, however, Dragon gives inverse characters for these ones. This is why you get the black space, and the inverted-video exclamation mark. Problems, problems!

●.K., we're in the problem-solving business here, so let's solve it. One way would be to ensure that only upper-case letter codes of 96 (denary) and above are used. You could do this in the BASIC part of the program by using lines like:

```
25 C=ASC(MID$(A$,N+1,1))
26IF C<64 THEN C=C+64
3Ø POKE&H7FEØ,C:NEXT
```

in place of the original line 3Ø. This solves the problem if you are making use of BASIC to put the phrase into place.

Another type of problem arises if you are making use of the variable list table. The program in Fig. 7.6 was rather wasteful in as much as the data for the text was stored twice – once as the variable A$, and again in the memory starting at S7FEØ. It would make more sense if we just used the variable list table entry. Getting the address for this is plain sailing because of the advanced BASIC that the Dragon uses. The VARPTR(A$) instruction, remember, gives the address of the first byte of a set which is the variable list table entry for AS. We can use this to pass these numbers to a machine code program, and make use of them. Figure 7.7 shows the flowchart. We shall find the start address for the text bytes, and the number of bytes (the length byte). We shall then follow the familiar pattern of loading the accumulator from the text list (the ASCII codes) and storing it at the screen. Each time round, we shall decrement the byte that represents the length of the string. When this byte is zero, that's an end to it. Figure 7.8 shows the assembly language program and the BASIC program that prepares the way. In this program, watch the difference between the two addressing methods for the registers. The X register is loaded from $7FE2, using extended addressing. This means that the numbers that are loaded into the X register are taken from addresses 7FE2 and 7FE3. It doesn't mean that the address $7FE2 is loaded into the X register, because that would

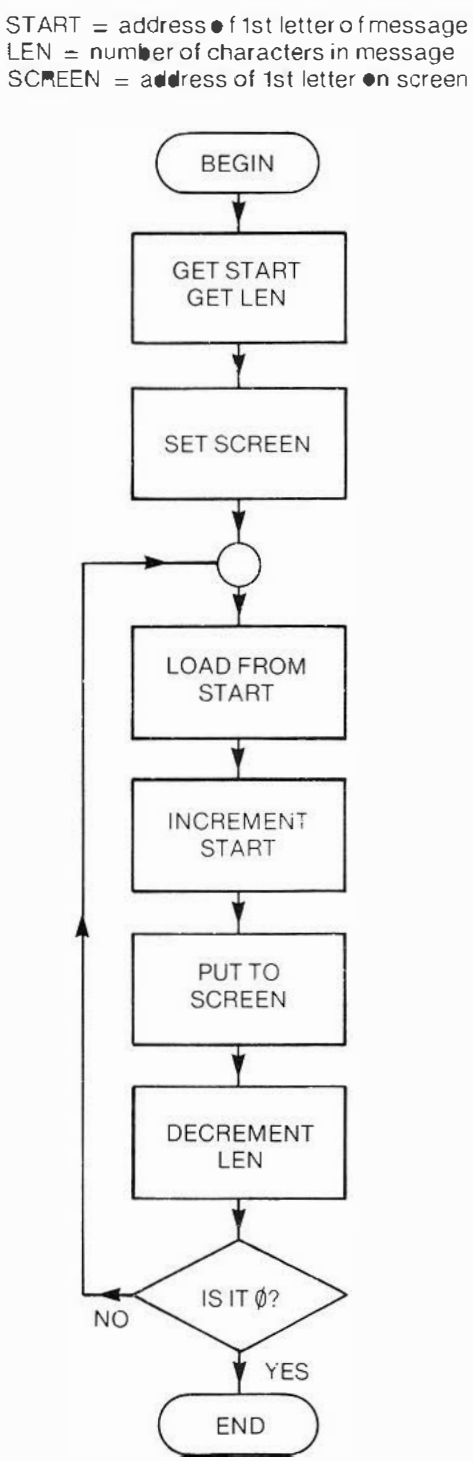


Fig. 7.7. The flowchart for a message program using VARPTR.

```
        LDX $7FE2     ; get address from 7FE2,
                        7FE3
        LDY #$Ø464    ; start of screen
LOOP:   LDA Ø,X+      ; load accumulator with
                        character, increment
        STA Ø,Y+      ; put it to screen, increment
        DEC #$7FEØ    ; decrement character count
        BNE LOOP      ; back if not zero
(a)     RTS           ; finished
```

```
10 CLEAR50,&H7F00:A$="DRAGON MAGIC!"
20 FOR J=0TO3:POKE &H7FE0+J,PEEK(VARPTR(
A$)+J)
30 NEXT:FORN=1TO17:READ D$
40 POKE&H7F00+N,VAL("&H"+D$):NEXT
50 CLS:EXEC&H7F01
100 DATABE,7F,E2,10,8E,04,64,A6,80,A7,A0
(b) ,7A,7F,E0,26,F7,39
```

Fig. 7.8. (a) The assembly language, and (b) the BASIC listing for the VARPTR message.

require immediate loading. The difference is important, because it's the address number that is stored in two bytes (in $7FE2 and $7FE3) that we want to be placed in the X register.

The result, once again, is the phrase, with a black square for a space, and with the inverted video exclamation mark! The reason is the same – we've put numbers directly into the screen memory, and the codes that serve perfectly well in a variable, or as CHR$() characters, give a different effect when used in this way. How do we sort out this one? The answer, as it so often turns out to be, is that the Dragon can take care of it! You see, all of these phrases look fine on the screen when you carry out a PRINT instruction in BASIC. There must be a routine, therefore, somewhere in the ROM of the Dragon, which converts the ASCII codes to the correct values for printing. If we can find the routine, then it's highly likely that we can use it for our own purposes.

Life is rather short to go wading through the whole ROM of the Dragon but, fortunately, this is a well-known routine. It starts at address $B54A, and it ends with a JSR, so we can call it as a subroutine any time we want it. Perhaps it might do what we want. The only snag is that this routine carries out the action of printing a character at the position of the cursor. Now we have deliberately chosen a place on the screen to print by using the address $Ø464 up

to now. How do we make the cursor move to this position? Once again, we need to know how Dragon does this. The answer is that two bytes in 'zero-page' memory, at $88 and $89 are used to keep the address of the cursor. If we put the address that we want to use in these places, we should then be able to call up the subroutine, and print what we want.

```
(a)         LDX $7FE2     ; get address
            LDY #$Ø464    ; screen memory
            STY $88       ; direct page ØØ88, cursor
      LOOP: LDA Ø,X+      ; load from X address,
                          increment
            JSR $B54A     ; PRINT subroutine
            DEC $7FEØ     ; length number decremented
            BNE LOOP      ; back if not zero
            RTS           ; finished
```

```
(b) 10 CLEAR50,&H7F00:A$="DRAGON MAGIC!"
    20 FOR J=0TO3:POKE &H7FE0+J,PEEK(VARPTR(
    A$)+J)
    30 NEXT:FORN=1TO21:READ D$
    40 POKE&H7F00+N,VAL("&H"+D$):NEXT
    50 CLS:EXEC&H7F01
    100 DATABE,7F,E2,10,8E,04,64,10,9F,88,A6
    ,80,BD,B5,4A,7A,7F,E0,26,F6,39
```

Fig. 7.9. (a) The assembly language, and (b) The BASIC listing for a program that makes use of the cursor address.

Figure 7.9 shows the result in assembly language and in BASIC form. Once again, we load the X register from the addresses $7FE2 and $7FE3. This will have the effect of placing the address of the start of the text into the X register. The screen address that we want to use is then loaded into the Y register. The third main step, however, is to store the address in the Y register into addresses $ØØ88 and $ØØ89. This is done by one instruction, which in assembly language is written as STY $88. This is a 'direct page addressed' instruction, which is shorter than an extended addressing method. We can use this because the direct page register is set to zero, so that specifying the number $88 in a direct page addresses instruction will automatically make use of the address $ØØ88. This has the effect of shifting the cursor to the correct position. We then load the first character of the message into the A register and call the printing routine at $B54A. When this returns, we decrement the character count in address 7FEØ, test to see if all of the characters have been

printed, and loop if they have not. When you try this one, you'll find that the words are correct, no dark spaces, no inverted video. Looks like another small step for a Dragon-user! If you're looking for problems, though, perhaps you might not want to have the cursor just following the phrase. Could you shift it? After all, if you can place it at address $Ø464 at the start of a program, you might be able to put it somewhere else at the end. You could try $Ø6ØØ, for example, or even addresses like $7FFA, which would make the cursor invisible. If you do tricks like this, though, you have to be careful that you return the cursor to the text screen when it is needed in a BASIC program.

Sailing out of the port

When I described the action of the computer system in Chapter 1, the idea of a *port* was raised. As far as a computer is concerned, a port is any chip or collection of chips that carries out the actions of sending bytes out or taking bytes in. As it happens, the port of the Dragon is rather a complicated one, and it is organised in a rather complicated way. For any of you who know something of the hardware of computer systems, there was an article by Mike James in *Electronics and Computing Monthly* for September 1983, which spilled a lot of the beans as far as this port is concerned. In this chapter, however, we'll concern ourselves with one action only – getting the Dragon to send out a sound signal.

What makes this complicated is that the port which the Dragon uses for this purpose is also used for other actions (cassette signals), and we have to 'configure' the port. This means set it up so that it can be used in the way that we want. The ways of doing this aren't so well known as some other aspects of Dragon machine code, and I'm indebted to Mike James, and also to Mr Opyrchal of Compusense for information on several methods, of which this is one. Another method is illustrated in Chapter 9.

To start with, Fig. 7.10 shows a BASIC poke program. This performs its setting up operations in lines 1Ø to 3Ø, and then pokes the address &HFF2Ø in a loop in lines 4Ø to 7Ø. This is the port address, and the setting up operations in the earlier lines have 'configured' the port correctly for sound output. What we're doing, then, is to place alternately high and low numbers into the port. This produces sound because circuits on the other side of the port convert these alternate high and low

```
10 POKE&HFF1D,&H34
20 POKE&HFF1F,&H35
30 POKE&HFF23,&H3F
40 FORN=1TO200
50 POKE&HFF20,128
60 POKE&HFF20,0
70 NEXT
```

Fig. 7.10. A BASIC program that pokes to the port to produce sound.

electrical voltages. These voltages are then sent out to the loudspeaker, and cause the sound. It's a fairly low sound, because BASIC is slow, and when the poke operations are performed slowly, the sound is a low sound. If we could perform these operations faster, we could obtain higher pitched sounds. The duration of the sound is given by the number of times we repeat the operations in the FOR...NEXT loop. When these pokes have been carried out, incidentally, you can hear the sound of cassettes being saved!

It's not so easy to carry out all this in assembly language. If we stick to setting up the port by using BASIC, we can produce a note in assembly language by using the routine shown in Fig. 7.11 in

```
        LDX #$1000  ; length of note number
LOOP:   LDA #$80    ; number for port, high
                      output
        BSR DELAY   ; port output
        LDA #$00    ; number for port, low
                      output
        BSR DELAY   ; port output
        LEAX -1,X   ; decrement X
        BNE LOOP    ; back if not finished
        RTS         ; finished
DELAY:  LDB #$80    ; pitch number
BACK:   STA $FF20   ; port output
        DECB        ; decrement pitch number
        BNE BACK    ; back if not zero
        RTS         ; back to main routine
```

Fig. 7.11. An assembly language routine for obtaining sound. The LDA #$00 could be replaced by CLRA, which is neater and needs only one byte.

assembly language form. The important feature of the assembly language version is that the number must be repeatedly sent to the port. It's not sufficient to send the number to the port and then have a time delay. While the time delay is counting down, you need to keep storing a number at the port address in each pass through the

counting loop. In the assembly language program of Fig. 7.11, the length of the note is determined by the number that is loaded into the X register at the start of the program. The main loop consists of loading the accumulator with $8Ø (high signal), sending this out a number of times and then loading $ØØ into the accumulator and sending this out also. The sending out and timing is carried out by the OUT routine. This loads $8Ø into the B register (this number decides what the pitch of the sound will be), and then loops round. In the loop, the byte that is in the A accumulator is sent to the port address of $FF2Ø, and the B register is decremented. This is repeated until the B register has reached zero. In the main loop, again, the X register is decremented by means of the LEAX −1,X step, and the main loop is repeated until this number also has been decremented to zero. That's it! Figure 7.12 shows the BASIC poke version of this,

```
10 CLEAR100,&H7F00:AD=&H7F00
20 POKE&HFF1D,&H34
30 POKE&HFF1F,&H35
40 POKE&HFF23,&H3F
50 FORN=1TO25:READ D$
60 POKE AD+N,VAL("&H"+D$)
70 NEXT
80 EXEC &H7F01
100 DATA 8E,10,00,86,80,8D,09,86,00,8D,0
5,30,1F,26,F4,39,C6,80,B7,FF,20,5A,26,FA
,39
```

Fig. 7.12. The listing of the BASIC poke program for sound.

which sounds a note for a fairly long time. The roughness of the note, like all notes that are created by the SOUND command in BASIC, is caused by 'interrupts'. The timer of the Dragon interrupts the action of the 6809 at intervals of a fiftieth of a second, and this places a 50 Hz signal over all sound outputs. If the interrupts are disabled, as they can be in machine code, the sound is very much better.

Now it would be much easier if we could make use of the ROM routine for this SOUND effect, but using the ROM routine in the Dragon is not quite so easy. We can, however, make some use of the addresses that the Dragon uses for sound. Address $ØØ8C is used to store the byte that determines pitch, and addresses $Ø Ø88 and $ØØ89 are used to store the duration numbers. We could, therefore, load up the B register in our program from $Ø Ø8C, and the X register from $ØØ88, and make use of these addresses to store whatever numbers we pleased to obtain a range of sounds. For now, though, this is as far as we go.

Chapter Eight

Debugging, Checking, DEMON and DASM

Debugging delights

Now that you have experienced some of the delights of machine code programming, it seems fair to mention some of the drawbacks. One of these is *debugging*. A 'bug' is a fault in a program, and debugging is the process of finding it and eliminating it. It all sounds rather insecticidal, but it's nothing like as easy as that!

It's easy to say, I know, but the first part is prevention. Check your flowchart carefully to make sure that it really describes what you want to do. When you are satisfied with the flowchart, turn to the assembly language to make sure that it will carry out the instructions of the flowchart. When you are happy with this, then check that the bytes you intend to poke into memory are the bytes which correspond to the assembly language instructions. One thing to watch very carefully is that you have the correct code for the addressing method that you are using. Another point which is peculiar to the 6809 is to ensure that you have the correct post-byte for any instruction that needs one. If you check each stage in the development of a program in this way, you will eliminate a lot of bugs before they are up and flying. Don't feel that you are a failure if the program still doesn't run – unless a machine code program is very simple, there's a very good chance that there will be a bug in it somewhere. It happens to all of us – and it's only by experience that you can get to the stage where the bugs will be few in number and easy to find.

If you use an assembler, one source of bugs completely disappears. Human frailty means that the process of converting assembly language instructions into machine code bytes is error-prone. That's because it means looking up tables, and anything which involves looking from one piece of paper to another is highly likely to introduce mistakes. I shall briefly describe the action of the

DASM assembler later in this chapter. At the time of writing, there were several assemblers available for the Dragon, but DASM has several advantages, one of which is that it can be obtained in one cartridge along with a monitor program called DEMON. There's more on monitors later in this chapter as well! If machine code has really caught your imagination, and you feel that you want to branch out into more advanced work than we have space for in this book, then a good assembler and monitor program is an essential. If, however, you intend to be just a dabbler, spawning the odd drop of machine code now and again, then the poke-to-memory methods that we have used so far will be perfectly adequate. Using these methods, however, means that there will be bugs lurking in each corner of the code. The main cause of these bugs is weariness. Converting an assembly language program into hex bytes, and writing them in the form of DATA lines for a BASIC poke program is a tedious job, and all tedious jobs result in mistakes (ever driven a 'Friday' car?). Faulty address methods are one common result of tedium, and simply writing down the wrong code is another. One very potent source of trouble is with branch displacements. You may get the number wrong somewhere between subtracting addresses and converting a number (particularly a negative number) to hex. Another problem arises when you modify a program, and add code between a jump instruction and its destination. Having done that, you then forget to alter the size of the displacement byte! This is a problem which simply doesn't arise when an assembler is used. An incorrect jump will nearly always cause the computer to lock up. You can often restore control with the RST button, but not always, and you will sometimes lose your program (you did record it, didn't you?). Another form of incorrect branch is doing the opposite of what you intended, like using BEQ in place of BNE or the other way round. Careful thought about what the jump will do for different sizes of bytes should eliminate this one.

A lot of problems, as I have already said, can be eliminated by meticulous checking, and it pays to be extra careful about branch displacements, and about the initial contents of registers. A very common fault is to make use of registers as if they contained zero at the start of the program. You can never be certain of this. It's safer, in fact, to assume that each register will contain a value that will drive the computer bananas if it is used. With all that said, and with all the effort and goodwill in the world, though, what do you do if the program still won't run?

There's no simple answer. It may be that your flowchart doesn't

do what you expect it to do, and if you didn't draw a flowchart, then you have got what you deserve. It may be that you are trying to make use of a Dragon ROM routine and it doesn't operate in the way that you expect. Until someone publishes a complete listing of the ROM routines, and the conditions (like what has to be in each register) that must exist when each is called, we are likely to find that this is a matter of trial and error. All I can do here is to give you general guidance on removing the bugs from a program that seems to be well-constructed but which simply doesn't work according to plan.

The first golden rule is never to try out anything new in the middle of a large program. Ideally, your machine code program will be made up from subroutines on tape, each of which you have thoroughly tested before you assembled them into a long program. In real life, this is not so easy, particularly when the subroutines exist only as DATA lines for BASIC poke programs. As usual, users of an assembler have the best of it, because they can keep assembly language instructions stored like BASIC programs, and merge and edit them as they choose.

The next best thing to keeping a subroutine library on tape is to have extensive notes about subroutines. In addition to routines of your own, you can keep notes on routines which you have seen in magazines. *Personal Computer World* runs a series called SUBSET (and I wish they would reprint it as a book!). This consists of several general-purpose machine code routines each month. Most of these are for the Z80, but the occasional 6809 routine creeps in. Even if you don't use the routines, the way in which they're documented should give you some ideas about how you should keep a record of your own routines – I personally would buy the magazine for this feature alone! If you are going to use a new routine in a program, it makes sense to try it on its own first so that you can be sure of what has to be in each register before the routine is called, and what will be in the registers afterwards. Look at the examples in SUBSET, and see how well this information is presented.

Planning of this type should eliminate a lot of bugs, but if you are still faced with a program that doesn't work, and which you don't want to have to pull apart, then you will have to use breakpoints. A breakpoint, as far as the Dragon operating system is concerned, is the byte $39. This is the RTS byte, and its effect is to return to BASIC. When you are back in BASIC, you can examine the contents of memory by using PEEK instructions. The principle is to pick a point in the program at which something is put into memory. If you place a $39 byte following this, then, when the program runs,

it will return to BASIC immediately after the $39 instruction is executed. By using a PEEK, you can then check that what has been loaded into the memory is what you expect. If it isn't, you should know where to look for the fault. If all is well at this point, then substitute the original byte that belongs in place of the $39, and place the $39 at the next place following a memory store command.

The most awkward fault to find by this or any other method is a faulty loop. A faulty loop always causes the computer to lock up. Though the RESET button will usually get you out of trouble, this will not always be the case. For example, it's possible for a program that runs wild to alter one of the bytes that controls the use of the keyboard, so that you find you can't use the keys even if you regain control! The main cause of this sort of thing is a loop back to the wrong position. For example, if we had a program, part of which read:

```
        LDB,#$FF
LOOP:DECB
        BNE LOOP
```

we could encounter problems. Suppose that this was assembled by hand, and we made the branch back to the LDB instruction rather than to the DECB instruction. This would result in the B register being kept 'topped up', and never decremented to zero, so that the loop would be endless. A mistake like this is easily spotted in assembly language, because the position of the label name is easy to check. It is very much more difficult to find when you have only the machine code bytes to look at. As always, taking care over loops is the only answer, and the method that has been shown in this book of calculating and checking displacements is a good precaution.

The DEMON monitor

I mentioned monitors briefly earlier on in this chapter. This has nothing to do with a TV monitor, which is a sort of superior quality TV display for signals. A monitor in the software sense is a program, one which checks (or monitors) each action of a machine code program. A monitor is (or should be!) a machine code program which can be put into the memory at a set of addresses that you aren't likely to use for anything else. Once there, a monitor allows you to display the contents of any section of memory (in hex), alter the contents of any part of RAM, and inspect or alter the register

contents of the 6809. These are the most elementary monitor actions, and it's useful if a section of program can be run, breakpoints inserted, and registers inspected on a working program. The ideal monitor would be one which could carry out the steps of a machine code program one at a time, displaying the register and memory contents at each step. Such a monitor was available for the old TRS-80 Mk. I, and would be a very welcome item for serious machine code programmers on other machines. There are rumours, as we go to press, of such a monitor becoming available for the Dragon.

Raising the DEMON

The DEMON monitor comes in cartridge form, and I used the combined monitor/assembler package known as DEMON/DASM. The Dragon has to be switched off while the cartridge is inserted, but once in place, the cartridge can be left in until you have need to use another type of cartridge. Simply placing the cartridge in position does not cause the monitor to run, you have to call it up by typing EXEC49154 (then press ENTER). This causes the DEMON menu to be displayed on the screen (Fig. 8.1).

Now you won't necessarily want to use all of these options – some of them might never be of particular interest to you. Instead of dealing with them in order, then, we'll pick a few that are the most useful to the beginner in machine code. Of these, the 'E' command (examine memory) is the most important. When you press E (don't use ENTER in any of these DEMON commands), the letter E appears at the bottom left-hand corner of the display where the cursor is flashing. You should then type the memory address that you are interested in. This must be typed as a four-digit hex number. In other words, if you want to see what is in $86, you must type ØØ86. You don't need to use $ or &H to mean hex, and you must not type any spaces between the E and the address. As you type the last digit, you will see the screen display a section of memory which includes the address that you have requested. The column at the left-hand side contains the starting address for each row of numbers, and there are eight numbers in each row. Since the numbers are in hex, the last digit of each address will be either a Ø or an 8. If, for example, you chose to look at ØØØØ, you would see address numbers on the left-hand side of the screen of ØØØØ, ØØØ8, ØØ1Ø, ØØ18, ØØ28 and so on, down to ØØ78. There are sixteen rows of eight numbers displayed,

Command	Meaning	Action
A	Alter registers	The cursor appears next to the CC register display. Typing a byte (two hex characters) will cause the cursor to move to the next register, the A register. At each register position, a byte (or two) can be entered, or BREAK pressed to move to the next register. Those values that are typed will be entered into the 6809 registers when a J or G command is used.
B	Break point	Type a four-digit hex address. A break point will be set at this address. When the program runs, it will stop at this address and display contents of registers.
E	Examine	Type a four-digit hex address. The screen will display memory contents starting at this address (or the nearest lower address).
F	Fill	Type two addresses and a data byte. Memory from first address to second will be fitted with the data byte.
G	Go	The program will be run starting at the address in the PC register.
J	Jump	Type address. The program will be run starting at that address.
M	Modify	Type memory address. The cursor displays the data byte which can be changed by typing two hex numbers. Several sub-commands available.
R	Register dump	Displays contents of 6809 registers.
V	Video page	Type two digits. This shifts text screen to a new address whose high byte has been typed; the low byte is ØØ.
X	Clear	Clear all break points and restore original bytes at these positions.
Z	Return	Return to BASIC.

Fig. 8.1. The DEMON menu – typical functions of a monitor.

which is enough to give you a good view of a chunk of memory. The numbers that are stored in the addresses are then shown in these rows. Suppose, for example, that we look at the row whose starting number (on the left-hand side) is $ØØ4Ø. The first byte in this row is the byte which is stored at $ØØ4Ø, the next byte is the one stored at $ØØ41 and so on.

While DEMON is displaying contents of memory in this way, there are four subsidiary commands that can be used. The down-arrow key will cause the next 'page' of information to be displayed, meaning the next sixteen rows of addresses. The up-arrrow key will display the previous sixteen rows of addresses. If you are displaying the first set (from $ØØØØ to $ØØ7F), then using the up-arrow will display the other end of memory, the page from $FF8Ø to $FFFF. The CLEAR key will cause the display to show either ASCII coded characters or hex codes. When you use the E command, at first, only hex codes are shown. Pressing CLEAR will cause any codes in the normal ASCII range to display as characters rather than in hex. Pressing CLEAR again will restore the normal display of hex codes again. This feature can be useful if you suspect that the section of memory that you are looking at contains messages or names of commands rather than machine code instructions. The fourth subcommand makes use of the @ key, and should be used only if you have a printer connected and switched on. Pressing the @ key in these circumstances will cause the information on the screen to be printed on the paper of the printer. If you have no printer connected, then the program will hang up if you have pressed @. You will have to press the RESET button to get back in control, then EXEC49154 to get back to DEMON. Pressing the BREAK key will get you back to the menu of DEMON, and this is true for any command – BREAK always causes a return to the menu.

The next most important instruction, as far as we are concerned at the moment, is 'R'. Pressing the R key causes the register contents of the 6809 to be displayed. Unless the machine has been interrupted in the middle of a program, there won't be much to look at here, but when you come to make use of the more advanced features of DEMON, like the use of breakpoints, it will be very useful. The contents of all the registers are shown, and normally we'll be looking at the contents of the A,B,X and Y registers in particular. You can press any key to return from this instruction.

In connection with these two instructions, we have the 'B' command. 'B' means 'set breakpoint', and when you type B you should follow it (no space) with a four-digit hex address. This is the

address at which the break byte will be put in, and when it has been entered, DEMON will return to its menu display. This break is not a simple return to BASIC, however, but a break to DEMON. What this means is that when you run your machine code program with the breakpoint inserted, the program will stop at the breakpoint, and the screen will show the contents of the registers, as if you had used the 'R' command. This is very often all that you need to spot where a program has gone wrong. DEMON allows you to set up to twelve breakpoints, pressing B, then entering the address each time. When the program stops at a breakpoint, you can inspect the contents of registers, use E to examine memory, and then allow the program to continue by pressing 'G'. You can even alter the contents of the registers before you start the program again, but this is not the sort of thing you want to try until you have rather more experience with machine code. It's useful, though, if you are checking the action of a loop, and you want to see what happens when the content of a register reaches some value like $FF or $ØØ. Instead of going round the loop dozens of times, you can go round once to check that the loop is working, and then alter the register contents so as to make the loop stop – and then check that it does. Altering the 6809 registers is done using the 'A' command of DEMON. Wherever you have used breakpoints, you should clear them after you have finished investigating, and the 'X' key performs this task.

These are very powerful methods of debugging and sorting out a program, and yet they form only part of the facilities that this very useful monitor offers. If, for example, you find that your machine code program is faulty, it can be mended by DEMON. Using the 'M' key brings up the 'alter memory' action. The M key has to be followed by a four-digit hex address, as usual, and causes a memory display to appear on the screen, like that of the 'E' command. The difference this time is that there is a flashing cursor over the first digit of the first byte. Now this allows you to change the byte. All you have to do is to type the new byte – two hex digits. If you only want to change the first byte, you still have to type two digits. For example, if you find 5F displayed, and you want 5A, you must type 5A. Despite what the manual says, you can't skip from the '5' to the 'F' and change only the 'F'. If you don't want to change a byte, you can skip to the next one (the next higher address, that is). This is done by using the right-arrow. You can move to the previous byte by using the left-arrow. You can move the cursor up or down a line by using the up- or down-arrow keys. The ENTER key will give you the next 'page' (16 lines of addresses), and the CLEAR key will give the

previous page. As usual, the BREAK key will return you to the menu.

Last, but quite certainly not least, the J key in the menu allows you to run a machine code program. This is something that you would want to do when you had made alterations or inserted breakpoints, or both. Following the J command, you type the start address of the program as a four-digit hex number. This will run the program, and if you have set any breakpoint, you will see the usual display of the register contents. When you have debugged the program to your satisfaction, you can then press the Z key to return to BASIC. One point, incidentally, is worth making here. When you first use DEMON after switching on the Dragon, or after using the RESET button, you call DEMON by using EXEC49154. When you leave DEMON temporarily to change a BASIC program, you should return to DEMON by using EXEC49156. This is a 'warm start' address, and it should be used when you are returning to DEMON as distinct from using it for the first time in that session. Using the 49154 address restarts everything, and can make it difficult to switch between DEMON and BASIC if you use it when you are returning to DEMON from BASIC.

Using the DASM assembler

At the time this book was being written, there were several assembler programs for the Dragon, but the DASM assembler was by a long way the most useful product in my estimation. Even though this is an introductory book for readers who may never go as far in machine code as to use an assembler, a description is necessary. Dispensing with a description of this assembler would be like writing a history of motoring and omitting the names of Rolls and Royce – even though not many readers would have direct experience of the product. You don't need Rolls Royce income levels to enjoy the use of DASM, in any case.

Any assembler worthy of the name will be written in machine code. DASM goes one step further by being in cartridge form. This is a great advantage, because an assembler that has to be loaded from tape can be a nuisance. The reason is that inevitably when you are developing a machine code program with an assembler, the program will run away from you at some stage in testing. When it does so, it usually manages to corrupt memory (change stored values), and it will be as likely to corrupt the assembler as anything

else if the assembler is in RAM. With the assembler in cartridge ROM, its code is safe, and you can return to it at any time.

All assemblers are different, and it's likely that my description of how to proceed with DASM will not suit the action of any other assembler. The principles, however, are the same, and it's a matter of learning a different sequence of commands. The differences between assemblers are rather like the differences between computers – but if you know the language, the differences become less important. The language in this case is 6809 assembly language. What you have to learn is how you type a program in such a form that the DASM assembler will deal with, process it into machine code, and store the code so that you can run the program.

Driving the DASM

Before you can get to grips with DASM, you need to know how it copes with the problem of assembling your instructions. The principle is that you write your assembly language program in numbered lines, just as you would write a BASIC program. This is not coincidental, because your program can be written even if the DASM cartridge is not installed! You are simply using the facilities of the Dragon to write lines of instructions. Provided you don't call on the Dragon to run them, you don't need the DASM present – not until you assemble the program, that is. Your lines of assembly language can be saved on tape in exactly the same way as you save any BASIC program (using CSAVE rather than CSAVEM, that is). What distinguishes this program from BASIC is that it uses assembly language, and that it contains directions to the assembler program.

Now the assembly language is fairly close to the standard that Motorola, the manufacturers of the 6809, have laid down. It isn't exactly identical. For example, some 6809 assembly language commands call for the use of square brackets, and the Dragon simply doesn't provide square brackets. Other differences are peculiar to the assembler program itself, and have made the program easier to write and use. We'll look at some of the differences here, but others are of interest only when you have had much more experience with machine code. The other way in which this program will differ from BASIC will be in the instructions to the assembler. It isn't enough to provide a set of assembly language instructions. You must specify in what addresses in memory you want them to be

assembled; whether you want to see the assembly language instructions appear on the screen as assembly takes place; whether you want to see error messages; whether you want output on paper (from the printer) or on the screen, and so on. As so often happens, though, you'll find that a standard set of these instructions will suffice for practically all of your uses. If you have no printer, for example, you don't need to pay any attention to the commands that affect the printer!

We would normally start, then, by clearing space. Clearing space means making enough string space for the assembly language, and enough memory space for the machine code. The ordinary BASIC instruction CLEAR is used here, because when you RUN a program that has been designed for the DASM assembler, the first part of it runs in BASIC. If you want to leave 100 bytes for strings, and allow all addresses above $7F00 to be used for machine code, then you will start your program with:

```
CLEAR 100,&H7F00
```

This allows you the addresses from $7F01 to $7FFF for machine code, and that's more machine code than you'll be writing for some time to come! Once you have cleared memory for your program in this way, that's an end to BASIC, and you have to make use of DASM for the rest of the work. The next line of your program must, therefore, start the assembler, and this is done by the line:

```
EXEC&HCFFA
```

The address &HCFFA is the start address for the DASM cartridge. Each instruction that follows this will therefore be dealt with by DASM and not by the BASIC ROM of Dragon. At the end of the assembly language program you need to use some method of signalling a return to BASIC (so that you can alter, record, etc.), and this is done by making the last line of the assembly language program the END instruction. This does not have to be the last line of all, because you can follow it with other instructions. These other instructions, however, will be instructions in BASIC to Dragon, not instructions in assembly language to DASM. The difference is important – BASIC can't make anything of assembly language, and DASM can't make anything of BASIC words.

Now as we go along, we'll look at refinements and useful additions to the stock of commands, but these are the fundamentals that you need to have firmly in your grasp in order to start making intelligent use of DASM. If each program you write starts with a CLEAR, then

uses EXEC&HCFFA, and ends with END, you are off to a flying start. Between these limits, you can write your assembler language. If you use a number with none of the distinguishing symbols in front of it (such as $), it will be taken as a denary number. Using $ means a hex number, and three other important marks are used. The exclamation mark, !, is used to put in ASCII code. If you type !A, then what is inserted is the ASCII code for A, which is 65 or $41. Another very useful mark is the asterisk, *. When this appears, it is taken to mean the 'current address'. This is the address of the start of the instruction in which the asterisk appears, so the asterisk is a useful way of indicating an address without actually having to know what it is! The asterisk is also used to indicate a line which is a comment, like the use of REM in BASIC. The other mark that is used extensively in programs for DASM is the @ sign. This is used to indicate a label name. Any word which starts with the @ sign will be used by DASM as a label word. If, for example, you type:

```
@LOOP LDA#$8Ø
```

then @LOOP is a label for the start of the LDA instruction. We can then put, later in the program, an instruction such as BEQ @LOOP.

Any assembler also permits what are called 'assembler directives', which are instructions to the assembler program itself. DASM is no exception, and some of these directives are worth looking at here because they (or versions of them) are used widely on assemblers for other microprocessors. We've already looked at END, which indicates the end of the assembly language. A directive which is found at the beginning of an assembly language program is ORG, which gives the address of the first byte of machine code. DASM doesn't, in fact, need to use ORG, because it places its code in the first free space that is reserved by the CLEAR instruction in BASIC. Nevertheless, ORG is available, and can be useful. If, for example, you want to assemble code at address $7FFØ, then you can type ORG $7FFØ (watch the space between the command word and the number – this space is essential). Another useful directive word is EQU. This, like ORG and END, does not cause any code to be assembled, but it defines a label. For example, you can type: @NOTE EQU $8Ø. and this will have the effect of making the assembler insert the number $8Ø everywhere it finds the label word @NOTE. Addresses can be specified in the same way, so that you can have @START EQU $7FØØ to define the label @START as $7FØØ, or you can use a line like @START EQU * to make the label word contain the address of the place at which the asterisk is placed

```
1Ø  CLEAR 1ØØ,&H7FØØ
2Ø  EXEC&HCFFA
3Ø  ALL
4Ø  LDX $7FE2
5Ø  LDY #$Ø464
6Ø  @LOOP LDAØ,X+
7Ø  STA Ø,Y+ : DEC#$7FEØ
8Ø  BNE @LOOP
9Ø  RTS
1ØØ END : EXEC
```

Fig. 8.2. A typical short DASM program, showing how this assembler is used.

Labels:	All label words *must* start with @. Any number of characters can be used, but only the first five following @, along with the last character, will be recognised.	
Symbols:	$	hex number follows
	!	ASCII code for following letter
	*	current address
	@	immediate address
	>	direct page address
Directives:	END	end of assembly language (return to BASIC)
	EQU	defines label
	RMB	reserves space for data
	FCB	generates data bytes (bytes follow FCB)
	FDB	generates address bytes
	FCC	generates string data
	●RG	defines start-of-program address
	DSP	display on monitor (number follows on control speed)
	PRT	print on printer
	OFF	suppresses printing and display
	ALL	display all instructions
	ERR	display only errors
	FML	full display of assembly language and code
	FMS	shortened version display
	PAG	set number of lines for display – press any key to continue
	PPO	print first pass – useful to locate some types of errors

Fig. 8.3. A summary of the DASM directives.

in the program. Figure 8.2 shows a short program in DASM form to illustrate how the directives are used, and Fig. 8.3 is a summary of the DASM directives.

This chapter, however, is not intended to give you a full description of the DASM assembler. What I intend is to give you a taste of what the use of an assembler can be like. If and when you are ready to use DASM, you will now be able to make sense of the (brief) instruction manual that comes with it. Even more important, you will be able to cope with other assemblers, and even with assemblers for other microprocessors, if you should ever change your machine.

Chapter Nine
Last Round-Up

One of the main problems in writing a book about machine code for beginners is knowing where to stop. Volumes could be written about the machine code programming of the Dragon and still leave room for more, so that any finishing point has to be an arbitrary one. My aim has been to introduce the subject and take you to a level at which you can start to make progress on your own. At this level, you can make use of the other books that are available, which treat machine code at a more advanced level. This chapter is concerned with tying up loose ends, mentioning a few more instructions, and illustrating how to make use of some features of the Dragon.

The stack

You can't get much further in machine code programming without coming across the word *stack*. A stack is a section of memory, and its special use is to preserve bytes that have been kept in registers. There's no special set of memory chips that we use as a stack – we can make use of any part of RAM that is not being used for something else. What you probably find difficult to understand at the present time is why we should ever need to use memory in this way.

Let's take a simple example. Suppose you have a program in which the X register is being used to hold a starting address of a number of bytes, and the Y register is being used to hold a starting address in the screen memory. This should be a familiar situation for you, because it's one that we have used extensively in previous chapters. Suppose, now, in the middle of this program, that we want to create a time delay by making use of a countdown in the X register. Whenever we load the count value into the X register, we shall have replaced the address that was stored there, and if we try to use the X register again in the rest of the program, we shall have to

re-load the address into it. This is what the stack is for. By means of a two-byte instruction, we can store the contents of one or more registers in the memory, and by using another similar instruction, we can get the values back into the correct registers again. The act of storing the register(s) on the stack is called 'pushing', and recovering the values is called 'pulling'. Any one or any group of registers can be 'pushed on the stack' and later 'pulled from the stack' by use of the two-byte commands of push and pull.

The 6809, however, has a small complication of two stacks. This doesn't mean that you have to use both of them, but it's another of the items that makes this such an excellent microprocessor from the point of view of the experienced programmer. One of the stacks is called the 'system stack', the other is called the 'user stack'. The idea is that if you have the 6809 in a computer system, with memory and ports, as we do in the Dragon, then this will use the 'system stack' for some of the operations of the 6809. These operations are ones that you don't have to program for yourself in detail. For example, if the action of the 6809 is interrupted by an electrical signal from a port (as it would be when an input arrived) it will carry out a routine called the 'interrupt service routine'. This means that it will automatically place the contents of all its registers on a stack, do whatever has to be done to deal with the interruption (this means carrying out a program that has been written for this purpose), and then return to normal service after recovering all the register contents from the stack. The stack that is used for this purpose is the 'system stack'. For your own routines, however, you can use the other stack, the user stack, which is also the stack that the Dragon uses for its own machine code routines. At this stage in our use of machine code, we will want to make only very limited use of any stack, and we would not normally want to shift the address that the Dragon uses for its stack. To use the user stack at another address, we would have to load into the U register (the user stack pointer) the address of the start of the memory that we wanted to use as stack. Start in this sense means the highest memory address of the memory that we are going to use. Suppose, for example, that we wanted to use the top of available memory of the Dragon 32 as user stack. We would have used CLEAR early in the program to make sure that this memory was not used by the system stack. We would then load an address such as $7FFF into the U register. From then on, we could use instructions such as PSHU (push on to user stack) and PULU (pull from user stack) to save the contents of registers while we made other use of these registers. The byte that follows the PSHU or

PULU byte then specifies which registers we want to 'save' in this way. All of the registers can be saved with the exception of the register we are using as stack pointer. For example, if you are using the U register for your stack, you can save all the other registers, including the S register, but not the U register. Figure 9.1 shows the post-bytes that have to be used to specify which registers are to be pushed or pulled.

Register	Post byte
PC	$8Ø
*S/U	$4Ø
Y	$2Ø
X	$1Ø
DP	$Ø8
B	$Ø4
A	$Ø2
CC	$Ø1

* S or U depending on which is not being used as a stock. To push/pull more than one register, add the post bytes. For example, to push/pull all registers, use post byte of $8Ø + $4Ø + $2Ø + $1Ø + $8 + $4 + $2 + $1 = $FF

Fig. 9.1. How the post-byte for register pushing or pulling is made up.

We'll look later at an example program which makes use of the stack but, for now, we're going to return to simpler matters. The rest of this chapter, in fact, will be devoted to examples of simple programs which will form a basis for developing into really useful routines for your Dragon. I must emphasise at this stage that you have now got to the launch-pad as far as machine code is concerned. From now on, what you need is practice, and all the information that you can lay your hands on. Look closely at every program for the Dragon that contains machine code, for example. Even if the machine code is in the form of bytes that are poked into memory, you can disassemble these by hand and find out what they do. By doing this, you can often discover addresses which will be very useful to you in your own programs. From now on, everything is potentially useful to you!

The KEYCHAR program

I'm going to take you through some steps in the development of a simple program. What I wanted to do was to make the keys of the Dragon give me graphics characters instead of letters. Since the letters use codes of Ø to 127, and graphics use codes of 128 upwards, the method was simple enough. All that I needed to do was to add 127 ($7F) to the code of a key, then place the new code into screen memory. Figure 9.2 shows the flowchart that I used to see what was

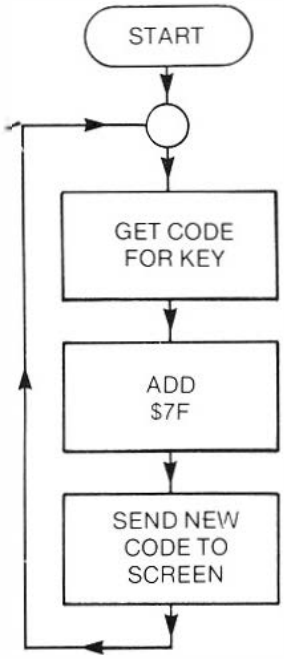


Fig. 9.2. A flowchart for the graphics keys program.

needed. Two ROM subroutines are needed. The one I have used at address $8ØØ6 is to place in the accumulator the code number for a key that is pressed – and this code will be Ø if no key is pressed. The other ROM routine is at $B54A, and this prints a character on the screen at the cursor position. Now it's possible that later versions of Dragon will have these routines at different addresses. Their addresses are, however, stored at other addresses! The address for the get-character routine is stored at $AØØØ and $AØØ1, and the address of the print-character routine is stored at $AØØ2 and $AØØ3.

Now this flowchart is not difficult to put into assembly language form. From here on, I'll show assembly language programs in the form that DASM uses, because the programs for this chapter were developed on DASM. If you're not using DASM, then you only need to convert the assembly languages of lines 5Ø onwards into

```
(a)  GET: JSR  $8ØØ6 ; get code for key
          BEQ  GET   ; back if zero = no key
          ADDA #$7F  ; add 127 denary
          JSR  $B54A ; display result
          BRA  GET   ; back for next one

(b)       BD 8Ø Ø6
          27 FB
          8B 7F
          BD B5 4A
          2Ø F4
```

Fig. 9.3. (a) Assembly language, and (b) hex codes for the graphics keys program.

machine code bytes, and poke these into memory in the way that you know by now. Figure 9.3 shows the assembly language and the code bytes for this program. We start by using the first subroutine for finding the code of a key. The BEQ step in line 6Ø then returns to the subroutine if the key-code is zero – meaning that no key is pressed. We then add $7F to the number in the accumulator to form a graphics character, and print this at the cursor position by using the second subroutine. So that we can create a pattern in this way, we then branch back to the start again to get another character.

Now this is a very simple routine, but it works very nicely. When you assemble the bytes into memory, and then use EXEC&H7FØl (just EXEC if you use DASM), you will see the cursor vanish because we have no cursor subroutine included in this program. When you press a key, however, you will see a graphics character appear. This is true for any of the keys, including the left arrow and the BREAK keys. Very interesting, that! It means that both the backspace and the BREAK keys give codes when they are pressed. The program has no way of escape, however, except by pressing the RESET button at the side of the Dragon. That's not really a very satisfactory state of affairs.

Now this is just the thing that you come across all the time in machine code programming. How can we improve the program? One obvious way is to provide a better way of getting back to BASIC! Suppose we could modify it so that we could return to BASIC when we pressed the BREAK key. To start with, we need to know how to find the code for the BREAK key. That doesn't need much research, just an INKEY$ loop in BASIC, such as:

```
1Ø K$=INKEY$:IF K$="" THEN 1Ø
2Ø ?ASC(K$)
```

If you run this, and press the BREAK key, you will find the number 3 printed. This is the code for the BREAK key, so if we can change our machine code program so as to detect this code, we can then cause a return to BASIC. What we need to do, then, is to include a CMPA #$Ø3 step immediately following the BEQ @GET step. In this way, if we have a number in the accumulator which is not zero, it will be compared with $Ø3, the code for the BREAK key. Next we need a BEQ step to cause a return to BASIC if this number is indeed $Ø3. This is performed by BEQ @OUT, and at the end of the program, beyond the BRA @GET step, we will place the @OUT label, with the instruction RTS. Figure 9.4 shows the complete

```
10 CLEAR100,&H7F00
20 EXEC&HCFFA
30 AL.L:FML
40 *KEYCHAR
50 @GET JSR $8006
60 BEQ @GET
70 CMPA #$03
80 BEQ @OUT
90 ADDA #$7F
100 JSR $854A
110 BRA @GET
120 @OUT RTS
130 END @GET
```

Fig. 9.4. The graphics keys program written for the DASM assembler.

program. Figure 9.5 shows the printout from the DASM assembly, which gives the addresses, codes and assembly language. It should give you an inkling as to why an assembler is such an essential item for the serious machine code programmer.

So far, so good. We can now type graphics characters, and return to BASIC when we have finished with them. After playing with this for a while, though, you begin to be irritated with the fact that you can't rub out any of the characters. The left-arrow key prints a character rather than rubbing out, and the reason is not hard to find. Whatever code number this left-arrow key provides has had $7F added to it before it has been placed on the screen. What we need is to detect this code, and make sure that the $7F is not added to it. How do we do that?

The answer is, just as we did before. We start by finding what code the key returns. We can use the BASIC inkey$ loop for this as before, and we come up with the number $Ø8 this time. We will therefore need a CMPA #$Ø8 step, and following it a BEQ, with a jump to the step that follows the ADDA#$7F. In this way, if the

```
7F01                        30      PRT
7F01                        40 *KEYCHAR
7F01 BD8006                 50 @GET JSR $80
06
7F04 27FB                   60      BEQ @GET

7F06 8103                   70      CMPA #$0
3
7F08 2707                   80      BEQ @OUT

7F0A 8B7F                   90      ADDA #$7
F
7F0C BDB54A                100      JSR $B54
A
7F0F 20F0                  110      BRA @GET

7F11 39                    120 @OUT RTS
7F12                       130      END @GET
```

Fig. 9.5. The printout from DASM, showing addresses and codes as well as instructions.

code that is detected is SØ8, then it is sent direct to the screen memory rather than having $7F added first. Figure 9.6 shows the result of this in assembly language. Now when we assemble and run the program, it gives graphics characters for all keys apart from BREAK and left-arrow. The BREAK key causes a return to BASIC, and the left-arrow key causes deletion, as it usually does. Now comes the challenge. What would you like this program to do, and what modification will you have to make? I've pointed the way, and shown you the methods – it's up to you now.

The KEYBEEP routine

Now we'll look at a routine that makes use of more advanced programming. The intention this time is to come up with a program that will cause a short beep to sound each time you press a key. This should happen when you are working normally in BASIC, so what we need is a way of inserting a piece of extra machine code into BASIC. This is quite a different matter from creating a machine code program that runs and then returns to BASIC, and we have to know rather more about the Dragon to be able to carry this out.

To start with, we have the problem of 'breaking in' to the routines that BASIC uses. Fortunately, the Dragon uses one particularly handy 'junction box'. What I mean by a junction box is a piece of code that is placed in the RAM rather than in the ROM. Any code

```
    10 CLEAR100,&H7F00
    20 EXEC&HCFFA
    30 ALL:FML
    40 *KEYCHAR
    50 @GET JSR $8006
    60 BEQ @GET
    70 CMPA #$03
    80 BEQ @OUT
    90 CMPA #$08
   100 BEQ @NOADD
   110 ADDA #$7F
   120 @NOADD JSR $B54A
   130 BRA @GET
   140 @OUT RTS
(a) 150 END @GET
```

```
7F01                        30      PRT
7F01                        40 *KEYCHAR
7F01 BD8006                 50 @GET JSR $80
06
7F04 27FB                   60      BEQ @GET

7F06 8103                   70      CMPA #$0
3
7F08 270B                   80      BEQ @OUT

7F0A 8108                   90      CMPA #$0
8
7F0C 2702                  100      BEQ @NOA
DD
7F0E 8B7F                  110      ADDA #$7
F
7F10 BDB54A                120 @NOADD JSR $
B54A
7F13 20EC                  130      BRA @GET

7F15 39                    140 @OUT RTS
(b) 7F16                   150      END @GET
```

Fig. 9.6. (a) The assembly language program that recognises the BREAK and left-arrow keys, and (b) the appearance of the DASM assembler printout.

that is placed in the RAM can be changed, unlike code in the ROM. The purpose of this, in fact, is to allow changes to be made by 'patching'. Patching in this sense means inserting a piece of your own program into a routine which is used by the operating system of the Dragon.

Looking for a place to patch is the most difficult part of any operation of this sort. There is one routine, located at $ØØA9 and $ØØAA, which can be used in this way, but it is executed only when

ENTER is pressed. This doesn't quite suit our needs, because for a KEYBEEP program, we want to break into a piece of program that is executed each time a key is pressed. If you don't have a listing of all the Dragon code (and I don't), then the only way to proceed is by some detective work. Here's what I did.

To start with, the operating system of the Dragon must make use of two routines that we already know about each time a key is pressed. One of these is the GETKEY routine, located at $8000 (in ROM), and the other is the print-character routine, located at $B54A. What we have to do is to look at the codes which start at these addresses. You can look at the codes in hex, but this is tedious. A much better idea is to use a program called a *disassembler*. This converts the codes into assembly language. There are no labels (though you can get labelling disassemblers for other micro-computers), and you will get peculiar results when the disassembler tries to read data, like reserved words. Nevertheless, any disassembler, even a very simple one in BASIC (see, for example, the article by Brian Cadge in *Your Computer*, May 1983) will be useful.

When a disassembler is used on these two sets of routines, the GETKEY routine doesn't look very promising. It has lots of JMP commands, but they all appear to be to addresses in the ROM. The other routine, however, starts with a subroutine call, JSR $0167. This is an address in the RAM, and it looks promising. It looks a lot more promising when we disassemble round this address. All the addresses from $015E to $01A8 contain the same byte, the $39 return-from-subroutine byte. This means that the whole of this piece of RAM behaves like a sort of telephone exchange. Many of the Dragon ROM routines must pass out to an address in this set, and then return. This is where we can do our patching.

What we shall do is this. At the address $0167, we shall replace the return byte $39 with three bytes. At $0167 we shall put a JMP byte, and in the next two addresses we shall place bytes of an address. This will be the address of a routine in RAM that we are going to patch in. If all is well, then, each time we press a key, our piece of machine code will be run before the computer can go back to normal. If we put the usual RTS byte at the end of our routine, then it will do exactly what the RTS in the location $0167 did – send the microprocessor back to the original routine. The routine that we're going to patch in is a simple one. It will prepare the port, using the simpler method of loading $BC into address $FF23, and then create a beep, using the sound subroutine that we looked at earlier. The port has to be loaded each time, because if it is not, then your

```
10 CLEAR200,&H7F00
15 POKE&HFF23,188
20 EXEC&HCFFA
30 ALL:FML
40 @JNCT EQU 359:@PORT EQU $FF20
45 @KEYBEEP EQU *
50 LDD #@START
60 STD @JNCT+1
62 LDA #$7E
64 STA @JNCT
70 RTS
80 @START PSHS X,B,A
90 LDX #$0F
100 @BEGIN LDA #$FC:BSR @BEEP
110 LDA #$00:BSR @BEEP
120 LEAX -1,X
130 BNE @BEGIN
135 PULS X,B,A
140 RTS
150 @BEEP LDB #$40
160 @BACK STA @PORT
170 DECB : BNE @BACK
180 RTS
190 END @KEYBEEP
```

(a)

Fig. 9.7. (a).

keybeeps will stop when there is any kind of error report, such as a syntax error.

Figure 9.7 shows the assembly language version of what we have come up with. This has been produced by the DASM assembler, so both the BASIC DASM lines, and the machine code are shown. If you are going to use this in the form of a POKE program, then you should use the bytes that start with the group CC7FØD at address 7FØ1, and make sure that they are poked to addresses that start at $7FØ1. If you want to use other addresses, the code must be altered. The critical part is the address $7FØD which occurs in the first line. If you change the address at which this subroutine starts, then the address of $7FØD must also be changed. It has to be the address at which the @START label is shown in the assembly language version. The program follows the lines that you might expect, but take particular heed of lines 11Ø and 18Ø. These make use of the system stack to save the contents of the X,A and B registers. This is because our subroutine will change the bytes in these registers, and it's highly likely that this would cause problems in the main routine. By saving the contents of these registers on the stack before the BEEP subroutine runs, and restoring them afterwards, we ensure that the BEEP routine does not interfere with the normal action. The system stack has been used because the user

```
7F01                        30      PRT
0167                        40 @JNCT EQU 35
9
FF20                        40 @PORT EQU $F
F20
7F01                        45 @KEYBEEP EQU
 *
7F01 CC7F0D                 50      LDD #@ST
ART
7F04 FD0168                 60      STD @JNC
T+1
7F07 867E                   62      LDA #$7E

7F09 B70167                 64      STA @JNC
T
7F0C 39                     70      RTS
7F0D 3416                   80 @START PSHS
X,B,A
7F0F 8E000F                 90      LDX #$0F

7F12 86FC                  100 @BEGIN LDA #
$FC
7F14 8D0B                  100      BSR @BEE
P
7F16 8600                  110      LDA #$00

7F18 8D07                  110      BSR @BEE
P
7F1A 301F                  120      LEAX -1,
X
7F1C 26F4                  130      BNE @BEG
IN
7F1E 3516                  135      PULS X,B
,A
7F20 39                    140      RTS
7F21 C640                  150 @BEEP LDB #$
40
7F23 B7FF20                160 @BACK STA @P
ORT
7F26 5A                    170      DECB
7F27 26FA                  170      BNE @BAC
K
7F29 39                    180      RTS
7F2A                       190      END @KEY
BEEP
```

(b)

Fig. 9.7. The assembly language for a keybeep routine that sounds a note whenever a key is pressed: (a) as typed for the DASM assembler and (b) the printout from the DASM assembler.

stack register was pointing to an unsuitable address, and there seemed to be no point in making the program longer by re-addressing the user stack.

When this program is run, and EXEC used, you will find that you get a beep when each key is pressed. If you don't, perhaps you forgot that the sound comes from the loudspeaker of the TV, and so the volume control of the TV has to be turned up! You'll find that the ENTER key causes a much longer beep, and that listings are also accompanied by beepings. It makes listings much slower also, because of the time delays that are built into the sound part of the routine. This could set you off on several new tracks. To start with, could you make a program which would assist a blind user, by giving a different note for each key pressed? It would mean using the code that is in the A register at the time when the registers are pushed to modify the 'pitch' byte that we put into the B register. Another track is that we could use this to have variable speed listings. Forget about the sound routine, just have the X register used as a time delay, taking its delay number from a spare piece of RAM. To slow down a listing, you would only have to do a poke to this RAM.

Video tricks unlimited

We have seen that we can use machine code to carry out some interesting and useful actions, but these have mainly concerned sound so far. It's time we looked at some of the possibilities that are presented by using machine code for video tricks. The trouble here is to know where to begin and, even worse, where to end. What you can do in this respect is limited only by your knowledge of how the video display of the Dragon operates, and by your curiosity in exploring what can be done.

Let's start, then, with a page-flicking trick. The Dragon uses a set of addresses from $FFC6 to $FFD3 to control the video start address. This means the address in memory of the byte that will control the appearance of the top left-hand corner of the screen. For the normal text screen, this address is $Ø4ØØ, for Page 1 it is $Ø6ØØ and so on – the addresses are given in your Dragon manual. The way this is controlled is rather elaborate, by poking 1's into these addresses from $FFC6 to SFFD3, and the table in Fig. 9.8 shows where the 1's must be placed to achieve different values of starting page address. What it amounts to is that it forms a seven-bit binary

```
START ADDRESS        POKE 1 to addresses (in hex)
Denary     Hex
   512     2ØØ       FFD2, FFDØ, FFCE, FFCC, FFCA, FFC8, FFC7
  1Ø24     4ØØ       FFD2, FFDØ, FFCE, FFCC, FFCA, FFC9, FFC6
  1536     6ØØ       FFD2, FFDØ, FFCE, FFCC, FFCA, FFC9, FFC7
  2Ø48     8ØØ       FFD2, FFDØ, FFCE, FFCC, FFCB, FFC8, FFC6

General Formula:  | A | B | C | D | E | F | G |   7 bit number (binary)

Multiply this number by 512 (denary) to get start address

A     Set : FFD3     Reset : FFD2 (store 1 for action)
B     Set : FFD1     Reset : FFDØ (store 1 for action)
C     Set : FFCF     Reset : FFCE
D     Set : FFCD     Reset : FFCC
E     Set : FFCB     Reset : FFCA
F     Set : FFC9     Reset : FFC8
G     Set : FFC7     Reset : FFC6
```

Fig. 9.8. The addresses that have to be poked to control the video addressing

number, whose value is then multiplied by 512 (denary) to give the start address.

Now we can get some interesting, but not always very useful, results by flicking from one page to another. You have probably achieved this in BASIC and, for graphics, the same can be done in machine code. The advantage of using machine code is that the flicking can be faster, but this is not necessarily an advantage, as we shall see. Something that you certainly can't do in BASIC, however, is to flick alternately between text and graphics. This looks like a good one to attempt with machine code.

To start with, let's see what we have to do. We shall have to put 1's into the right addresses, to start with. For a video display that starts on the first high resolution page, we need to put the binary number 11 into the memory positions. This means putting a 1 into SFFC7, and a 1 into $FFC9. We shall then need a time delay, to hold this picture on the screen, and then we can change the number to 1Ø by placing a 1 into address SFFC6. If you can't see why we're using these numbers, then 1Ø is denary 2, and 2*512 is 1Ø24, the address of the start of the text screen. Binary 11 is 3, and 3*512 is 1536, the start of page 1 of high resolution graphics. Figure 9.8 then shows where we have to place 1's to achieve these numbers 11 and 1Ø.

This, however, isn't enough, and we have to change the numbers in another memory address. This one is $FF22. It must contain Ø when the text screen is being used, and $CØ when the high resolution PMODE1 screen is being used. How did I find that one? Trial and

error – and it took a lot of trying. Once these are correctly placed, though, we should be able to swap pages. We shall then need a time delay so that the swapping isn't too fast. That sounds like fairly straightforward programming, and Fig. 9.9 shows the result.

After the usual CLEAR command, the GOSUBIØØ in line 2Ø carries out the assembly of the machine code. This, once again, is in DASM form. If you are assembling by hand, then the subroutine at line IØØ will consist of a loop that will POKE the code numbers into memory, starting at &H7FØI, just as we have done earlier. When the subroutine returns, a box is drawn on the high resolution screen, and then line 5Ø starts the flicking by calling the machine code routine. The flicking is comparatively slow, and there is a reason for this. If you attempt to flick fast, you will run into display difficulties. The TV display consists of a dot moving across and down the screen, and the whole action is repeated at 25 complete pictures per second. If your flicking approaches this speed, there will be unpleasant 'interference' effects. You can imagine, for example, what could happen if you switched from one page to another while the spot was halfway down the screen. Fast flicking, unless you can find just the

```
10 CLEAR100,&H7F00
20 GOSUB100
30 PMODE1:COLOR2,3:PCLS
40 LINE(10,10)-(140,180),PSET,BF
50 EXEC&H7F01
60 END
100 EXEC&HCFFA
110 @BEGIN PSHS X,Y
120 LDX #$FFC6
130 @SWITCH LDA #$01:STA 1,X
140 STA 3,X
144 LDA #$C0:STA $FF22
150 BSR @DELAY
160 STA 0,X
164 LDA #$00:STA $FF22
170 BSR @DELAY
180 JSR $8006
190 CMPA #$03
200 BNE @SWITCH
210 PULS X,Y
220 RTS
230 @DELAY LDY #$4FFF
240 @LOOP LEAY -1,Y
250 BNE @LOOP
260 RTS
270 END @ BEGIN
280 RETURN
```

(a)

```
7F01                    105     PRT
7F01 3430               110 @BEGIN PSHS
X,Y
7F03 8EFFC6             120     LDX #$FF
C6
7F06 8601               130 @SWITCH LDA
#$01
7F08 A701               130     STA 1,X
7F0A A703               140     STA 3,X
7F0C 86C0               144     LDA #$C0

7F0E B7FF22             144     STA $FF2
2
7F11 8D13               150     BSR @DEL
AY
7F13 A784               160     STA 0,X
7F15 8600               164     LDA #$00

7F17 B7FF22             164     STA $FF2
2
7F1A 8D0A               170     BSR @DEL
AY
7F1C BD8006             180     JSR $800
6
7F1F 8103               190     CMPA #$0
3
7F21 26E3               200     BNE @SWI
TCH
7F23 3530               210     PULS X,Y

7F25 39                 220     RTS
7F26 108E4FFF           230 @DELAY LDY #
$4FFF
7F2A 313F               240 @LOOP LEAY -
1,Y
7F2C 26FC               250     BNE @LOO
P
7F2E 39                 260     RTS
7F2F                    270     END @ BE
(b)  GIN
```

Fig. 9.9. (a) The assembly language as typed for DASM, and (b) the printout from the DASM assembler for a page-swapping program.

right speed, will therefore cause pictures that appear broken up, just as if the TV was maladjusted.

Looking now at the assembly language, we begin (line 11Ø) by pushing the X and Y registers on to the stack. This is a precaution in case these registers are being used by the main program, as is likely. Lines 13Ø to 144 then place the correct bits and bytes into memory to activate the high resolution screen. Line 15Ø then calls up the delay,

and lines 160 and 164 cause the switch to the text screen again. After another delay, the keypress routine in ROM is called, and tested to see if the BREAK key is pressed. This, you remember, causes a $03 to appear in the A register. If the BREAK key is pressed, the registers are restored, and the program returns to BASIC. If the BREAK is not pressed, the loop repeats. It's a feature of machine code programs that if you want to include a feature like stopping with the BREAK key, then you have to program it – it doesn't happen automatically!

Blob-chaser's delight

When all's said and done, though, a lot of machine code effort goes into games, and we'll end up by looking at the sort of thing that is in greatest demand – an object moving on the screen. Now if we were to explore every possibility of movement from descending aliens to leaping frogs, we would need a book the size of *War and Peace*. We have already broken the ground for this topic in earlier chapters, where we dealt with the video screen addresses, and how we could form shapes by poking numbers into memory. What we need to look at now, then, is how we would move a shape around. The advantage of using machine code for this type of job is that the speed of BASIC is no longer a limitation.

In accordance with the rule I have followed throughout – keep it simple – I'll look at a simple move-a-blob program. What we shall do is to define the memory we are going to use, with the ordinary BASIC high resolution graphics commands. We shall then see how this blob can be moved about in the memory, and so on the screen. Everything else is a variation on this, believe it or not. The idea is that we take the start of a page of screen memory, and place its address into the X register. We then load the accumulator with the byte we want to display, and store it at the address in the X register. By using X-indexed storing, with auto increment, we shall ensure that the address in the X register increments after the store operation. We then call a time delay, and blank out the image. Since the address has been incremented, this has to be done by using a −1 in the indexed instruction. Another delay follows, and then the loop is repeated. We can bring the program to an end by comparing the address in the X register with the end address of the video memory for the PMODE that we have used.

Figure 9.10 shows the program in its BASIC plus DASM form,

```
10 CLEAR100,&H7F00:CLS
20 PMODE1:SCREEN1,0
30 GOSUB100:PCLS
40 EXEC&H7F01
50 GOTO50
100 EXEC&HCFFA
110 @START LDX #$600
120 @LOOP LDA #$0F
130 STA 0,X+
140 BSR @DELAY
144 LDA #$00:STA -1,X
146 BSR @DELAY
150 CMPX #$11FF
160 BNE @LOOP
170 RTS
180 @DELAY LDY #$05FF
190 @BACK LEAY -1,Y
200 BNE @BACK
210 RTS
220 END @START
230 RETURN
```

Fig. 9.10. A blob-moving program in assembly language form, as written for the DASM assembler.

and Fig. 9.11 shows the printout from DASM, which gives you the codes for a POKE program. By this time, you should be able to follow what is going on in the program with no difficulty. What is not so easy is to see where we go from there. Well, here's your starter for ten. If you make the X register increment by the number of address positions in a line, the blob will appear to move vertically. You can do that by using the command LEAX 8Ø,X (for example) in place of the auto increment. The effect of LEAX 8Ø,X will be to add 8Ø (denary) to the value in X each time it is used. You then have to arrange it so that the byte is stored, and after a time delay is wiped. The LEAX step then moves you to the next vertical position, and the process can be repeated. In each case, you can use STA Ø,X rather than the X+ that was needed before. How do you move an object that consists of several bytes? The answer is that you keep the bytes in a table, and you carry out the storing and the wiping operations in a loop each time. There's very little that's new to learn, except about the hidden surprises that the Dragon keeps for you. As you go on, though, you will accumulate experience, until you find that the surprises are fewer, and you can find ways round them more easily. When that time arrives, you're entitled to call yourself an expert, a St George of the computing world!

```
7F01                    105      PRT
7F01 8E0600             110 @START LDX #
$600
7F04 860F               120 @LOOP LDA #$
0F
7F06 A780               130      STA 0,X+

7F08 8D0C               140      BSR @DEL
AY
7F0A 8600               144      LDA #$00

7F0C A71F               144      STA -1,X

7F0E 8D06               146      BSR @DEL
AY
7F10 8C1 1FF            150      CMPX #$1
1FF
7F13 26EF               160      BNE @LOO
P
7F15 39                 170      RTS
**7F16 108E05FF           180 @DELAY LDY
 #$05FF
7F1A 313F               190 @BACK LEAY -
1,Y
7F1C 26FC               200      BNE @BAC
K
7F1E 39                 210      RTS
7F1F                    220      END @STA
RT
```

Fig. 9.11. The DASM printout, showing the hex codes.

Appendix A

How Numbers Are Stored

The Dragon uses five bytes of memory to store any number. This Appendix describes how numbers are stored, but if you have no head for mathematics, you may not be any the wiser!

To start with, numbers are stored in *mantissa-exponent* form. This is a form that is also used for denary numbers. For example, we can write the number 216000 as 2.16×10^5 or the number .00012 as 1.2×10^{-4}. When this form of writing numbers is used, the power (of ten in this case) is called the exponent, and the multiplier (a number greater than 1 and less than 1) is called the mantissa. Binary numbers can also be written in this way, but with some differences. To start with, the mantissa of a binary number that is written in this form is always fractional, but no point is written. Secondly, the exponent is a power of two rather than a power of ten. We could therefore write the binary number 1011000000 as 1011E1000. This means a mantissa of 1011 (imagine it as .1011) and exponent of 1000 (2 to the power 8 in denary). There's no advantage in writing small numbers in this way, but for large numbers, it's a considerable advantage. The number:

11010100000000000000000000

for example can be written as 11010101E11000 (think of it as .11010101 $\times 2^{24}$).

This scheme is adapted for the Dragon, and other machines which use Microsoft BASIC. Since the most significant digit of the mantissa (the fractional part of the number) is always 1 when a number is converted to this form, it is converted to a 0 for storage purposes. The exponent has (denary) 128 added to it before being stored. This allows numbers with negative exponents to be stored without complications, since a negative exponent is then stored as a number whose value is less than 128 denary. The Dragon uses four bytes to store the mantissa of a number, and one byte to store the exponent.

To take a simple example, consider how the number 20 (denary) would be coded. This converts to binary as 1010, which is .10100000 $\times 2^5$, writing it with the binary point shown, using eight bits, and with the exponent in denary form. The msb of the fraction is then changed to 0, so that the number stored is 00100000. Peeking this memory will therefore produce the number (denary) 32 in the mantissa lowest byte. Meantime, the exponent of 5 is in binary 101. Denary 128 is added to this, to make 10000101. Peeking this memory will give you 133 (which is 128+5).

Appendix B

Assemblers, Disassemblers and Monitors

An assembler is a program, on cassette or in cartridge form, which will 'assemble' instructions that are in assembly language into machine code. The machine code can then be recorded (using CSAVEM) and used. A disassembler performs the opposite action of operating on machine code to produce assembly language. A simple disassembler, however, cannot distinguish between machine code instructions and other bytes (such as data), nor can it add labels to the disassembled listing. A good disassembler for the serious machine code programmer should allow hard copy (a disassembly on the screen does not show enough of the listing to follow easily), and should separate data from code. A listing disassembler is the height of luxury. A monitor permits the contents of memory and the 6809 registers to be kept under surveillance, and can also be used in editing and debugging programs.

There are several assemblers that are available for the Dragon at the time of writing. The official Dragon assembler, DREAM, had not appeared at the time of writing, and I used the excellent DEMON/DASM package that is described in Chapter 8. This is available from Compusense (see magazines for the address and current prices). As a disassembler, I used the BASIC program first written by Brian Cadge, and published in *Your Computer* in the May 1983 issue. A modified version of this program is included here as Appendix H. I am grateful to the author and to the Editor of *Your Computer* for permission to reprint this listing, with my modifications.

Appendix C

Hex and Denary Conversions

(a) Hex to Denary

For single bytes (two hex digits) –
Multiply the most significant digit by 16, and add the other digit.
For example:

$3D is 3*16 + 13 =61 denary.

For double bytes (address numbers) –
Write down the least significant digit. Now write under it the value of the next digit, multiplied by 16. Under that, write the next digit, multiplied by 256. Under that, write the next digit, multiplied by 4Ø96.
For example:

$F3DB converts as follows:

Write Is digit	11
Next digit*16 is 13*16	2Ø8
Next digit*256 is 3*256	768
Next digit*4Ø96 is 15*4Ø96	61440
Now take the total, which is	62427

Denary to hex

For single bytes (less than 256 denary) –
Divide by 16. The whole part of the number is the most significant digit. The least significant digit is the fractional part of the result multiplied by 16.
For example:

To convert 155 to hex:
153/16 = 9.6875, so 9 is the most significant digit.
The least significant digit is .6875 * 16, which is 11. This converts to hex B, so that the number is $9B.

For double-byte numbers (numbers between 256 and65535 denary)
Divide by 16 as before. Note the whole number part of the result, and write down the fractional part, times 16, as a hex digit. Repeat the action with the whole number part, until only a single hex digit remains.
For example:

To convert 23815 to hex:
23815/16 = 1488.4375. The fraction .4375*16 gives 7, and this is the least significant digit.
Taking the whole number part, 1488/16=93.ØØ. Since there is no fraction, the next hex digit is Ø.
93/16=5.8125. The fraction .8125, multiplied by 16 gives 13, which is hex D. This is the third hex figure. Since the whole number part is less than 16 (it's 5), then this is the most significant digit, and the whole number is $5DØ7.

Appendix D

The Instruction Set

The instruction set of the 6809 is fairly large, and there will be many instructions in this list which you may never need to use unless you go in for very advanced programming indeed. A full description of the action of each instruction would take too much space, and so the action has been indicated by abbreviations. In general, M means a byte at an address in memory, and the registers are referred to under their usual letter references. An arrow indicates where the result of an action is stored. For example, A+M→A means that the byte in the memory (addressed by the instruction) is added to the byte in the accumulator A, and the result is placed in the accumulator A. When the double-byte registers are used in this way, two memory addresses are needed. This has been indicated as M:M+1. This means that the more significant byte is stored at the addresses memory M, and the less significant byte at the next memory address, M+1. Where an instruction needs a post-byte, this has been written as PB except for indexed addressed operations. All indexed addressed operations will require a post-byte.

The instruction codes are shown in columns graded by the addressing method. These methods are Immediate, Direct Page, Indexed, Extended, and Inherent. The Inherent addressing method means that no special addressing is needed. C means the carry bit, CC means the condition-code register. D is the double register formed from A and B. All codes are in hex. For shifts, A has been used to mean arithmetic shift, and L, to mean logic shift (see text for details).

Form	Imm.	D.P.	Indx.	Extd.	Inh.	Description
ABX	—	—	—	—	3A	B+X→X
ADCA	89	99	A9	B9	—	A+M+C→A
ADCB	C9	D9	E9	F9	—	B+M+C→B
ADDA	8B	9B	AB	BB	—	A+M→A
ADDB	CB	DB	EB	FB	—	B+M→B
ADDD	C3	D3	E3	F3	—	D+M:M+1→D
ANDA	84	94	A4	B4	—	A AND M→A
ANDB	C4	D4	E4	F4	—	B AND M→B
ANDCC	1C	—	—	—	—	CC AND M→CC
ASLA	—	—	—	—	48	A. shift left A
ASLB	—	—	—	—	58	A. shift left B
ASL	—	Ø8	68	78	—	A. shift left mem.
ASRA	—	—	—	—	47	A. shift right A
ASRB	—	—	—	—	57	A. shift right B
ASR	—	Ø7	67	77	—	A. shift right mem.
BITA	85	95	A5	B5	—	A AND M,change CC
BITB	C5	D5	E5	F5	—	B AND M,change CC
CLRA	—	—	—	—	4F	Ø→A
CLRB	—	—	—	—	5F	Ø→B
CLR	—	-ØF	6F	7F	—	Ø→M
CMPA	81	91	A1	B1	—	M−A,setCC
CMPB	C1	D1	E1	F1	—	M−B,set CC
CMPD	1Ø83	1Ø93	1ØA3	1ØB3	—	M:M+1−DD, set CC
CMPS	118C	119C	11AC	11BC	—	M: M+1−S, set CC
CMPU	1183	1193	11A3	11B3	—	M:M+1−U, set CC
CMPX	8C	9C	AC	BC	—	M:M+1−X, set CC
CMPY	1Ø8C	1Ø9C	1ØAC	1ØBC	—	M:M+1−Y, set CC
COMA	—	—	—	—	43	invert A
COMB	—	—	—	—	53	invert B
COM	—	Ø3	63	73	—	invert M
CWAI	3C	—	—	—	—	CC AND M, wait.
DAA	—	—	—	—	19	Decimal adjust A
DECA	—	—	—	—	4A	A−1→A
DECB	—	—	—	—	5A	B−1→B
DEC	—	ØA	6A	7A	—	M−1→M
EORA	88	98	A8	B8	—	A EOR M→A
EORB	C8	D8	E8	F8	—	B EOR M→B
EXG	1E	—	—	—	—	R1⇄R2 P Bexchange
INCA	—	—	—	—	4C	A+→A
INCB	—	—	—	—	5C	B+1→B
INC	—	ØC	6C	7C	—	M+1→M
JMP	—	ØE	6E	7E	—	Jump to address

Form	Imm.	D.P.	Indx.	Extd.	Inh.	Description
JSR	—	9D	AD	BD	—	Jump to subroutine
LDA	86	96	A6	B6	—	M→A
LDB	C6	D6	E6	F6	—	M→B
LDD	CC	DC	EC	FC	—	M:M+1→D
LDS	10CE	10DE	10EE	10FE	—	M:M+1→S
LDU	CE	DE	EE	FE	—	M:M+1→U
LDX	8E	9E	AE	BE	—	M:M+1→X
LDY	108E	109E	10AE	10BE	—	M:M+1→Y
LEAS	—	—	32	—	—	M→S , PB
LEAU	—	—	33	—	—	M→U , PB
LEAX	—	—	30	—	—	M→X , PB
LEAY	—	—	31	—	—	M→Y , PB
LSLA	—	—	—	—	48	L. left shift A
LSLB	—	—	—	—	58	L. left shift B
LSL	—	08	68	78	—	L. left shift M
LSRA	—	—	—	—	44	L. right shift A
LSRB	—	—	—	—	54	L. right shift B
LSR	—	04	64	74	—	L/right shift M
MUL	—	—	—	—	3D	A*B→D unsigned
NEGA	—	—	—	—	40	−A→A
NEGB	—	—	—	—	50	−B→B
NEG	—	00	60	70	—	−M→M
NOP	—	—	—	—	12	No operation
ORA	8A	9A	AA	BA	—	A OR M→A
ORB	CA	DA	EA	FA	—	B OR M→B
ORCC	1A	—	—	—	—	CC OR M→CC
PSHS	—	—	—	—	34	Push to S stack, PB
PSHU	—	—	—	—	36	Push to U stack, PB
PULS	—	—	—	—	35	Pull from S stack, PB
PULU	—	—	—	—	37	Pull from U stack, PB
ROLA	—	—	—	—	49	rotate left A
ROLB	—	—	—	—	59	rotate left B
ROL	—	09	69	79	—	rotate left M
RORA	—	—	—	—	46	rotate right A
RORB	—	—	—	—	56	rotate right B
ROR	—	06	66	76	—	rotate right M
RTI	—	—	—	—	3B	Return from interrupt.
RTS	—	—	—	—	39	Return from subroutine.
SBCA	82	92	A2	B2	—	A−M−C→A
SBCB	C2	D2	E2	F2	—	B−M−C→B
SEX	—	—	—	—	1D	Extend sign B→A
STA	—	97	A7	B7	—	A→M

Form	Imm.	D.P.	Indx.	Extd.	Inh.	Description
STB	—	D7	E7	F7	—	B→M
STD	—	DD	ED	FD	—	D→M:M+1
STS	—	10DF	10EF	10FF	—	S→M:M+1
STU	—	DF	EF	FF	—	U→M:M+1
STX	—	9F	AF	BF	—	X→M:M+1
STY	—	109F	10AF	10BF	—	Y→M:M+1
SUBA	80	90	A0	B0	—	A−M→A
SUBB	C0	D0	E0	F0	—	B−M→B
SUBD	83	93	A3	B3	—	D−M:M+1→D
SWI1	—	—	—	—	3F	Software interrupt 1
SWI2	—	—	—	—	103F	Software interrupt 2
SWI3	—	—	—	—	113F	Software interrupt 3
SYNC	—	—	—	—	13	Synchronise to interrupt
TFR	—	—	—	—	1F	Exchange content, P B
TSTA	—	—	—	—	4D	A−0, set CC
TSTB	—	—	—	—	5D	B−0, set CC
TST	—	0D	6D	7D	—	M−0, set CC

Register exchange post-byte

The byte is formed from two halves. The upper half is a code for the source register, the lower half for the destination register. A four-bit code is used to indicate the registers. The codes are as follows:

Register	Code
D	$0
X	$1
Y	$2
U	$3
S	$4
PC	$5
A	$8
B	$9
CC	$A
DP	$B

For example, if the source register was X and the destination U, then the upper byte would be $1, and the lower one 3, making the post-byte $13.

Appendix E

Addressing Methods of the 6809

Each addressing method has the effect of using a byte in the memory. The address at which this byte is stored is called the Effective Address (EA). The purpose of any addressing method is to make use of an effective address.

Immediate Addressing: The EA is the address that immediately follows the instruction byte.

Direct Page (DP) Addressing: Only the lower byte of the EA is given in the instruction. The upper byte is contained in the DP register. The DP register of the Dragon usually contains $\$\emptyset\emptyset$, so that an instruction using DP addressing with a byte of FB would make use of the address $\$\emptyset\emptyset$FB.

Indexed Addressing: A number is stored in one of the index registers (X or Y). The effective address is this number plus any displacement that is specified in the instruction. For example, LDA 5,X means load the accumulator from the address which consists of the number stored in the X register, plus 5.

Extended Addressing: The instruction is followed by two bytes, which form a complete address. For example, LDA $563F means that the accumulator is to be loaded from the address $563F.

Inherent Addressing: The address is implied by the instruction, and no special address reference is needed. For example, CLRB means clear (zero) the B register, and no EA is needed.

Indirect Addressing: Several of the addressing methods can be used indirectly. This means that the addressing method finds two bytes. These in turn form the effective address. Indirect addressing should be avoided by the beginner until its capabilities are well understood. For that reason, it has not been used in this book.

Appendix F

A Few ROM and RAM Addresses

Until a full disassembly of the Dragon ROM is available, not many of the ROM addresses will be widely known. A selection of the most useful addresses is shown below. Along with this, I have included some useful RAM locations, with brief notes on what is stored there.

ROM Addresses

$8ØØ6
This scans the keyboard, and places the value found into the A register. A=Ø if no key is pressed.

$B54A
This prints on the screen the character which is in the accumulator.

$8Ø2I
This starts the motor of the cassette recorder.

$B93E, $B999 and $8Ø IB
These are all concerned with reading and writing tapes. They should not be used by the beginner.

RAM locations (All addresses in hex)

$17, $18	BASIC stack address.
$19, $1A	Start of BASIC address.
$1B, $1C	Start of simple variable address.
$1D, $1E	Start of arrays address.
$1F, $2Ø	Start of free space address.
$21, $22	Bottom of string space address.
$23, $24	Address of next available byte in string space.
$25, $26	Address of start of last string used.
$27, $28	Address of top of string space.
$2F, $3Ø	Address of current line.
$33, $34	Address of next byte of DATA.
$35, $36	Address of next byte in keyboard buffer.
$74, $75	Address of top of memory, less 1.
$87	ASCII code of last key pressed.
$88, $89	Address of cursor in memory.
$8C	Store pitch byte for SOUND.
$8D, $8E	Store pitch*duration number.
$9D, $9E	Address for EXEC.
$A6, $A7	Address of BASIC program portion being executed.
$Ø2DD-$Ø3D4	Keyboard buffer space.

Appendix G
Magazines and Books

The problem of where you go from here is solved by looking at the magazines and books that are available. The groundwork that this book has supplied should allow you to go on to any of the books that deal with 6809 programming, but which are not very useful to the complete beginner. A look at the books available in your local Computer store, or from suppliers such as Mine of Information (in St Albans) will show you what is available.

Magazines are also a fruitful source of ideas. *Personal Computing World's* SUBSET series is a very valuable source of ideas in machine code programming. *Your Computer* frequently prints articles on machine code topics for the Dragon, and you should watch out for listings which may reveal the use of ROM routines that you haven't met before. Remember that a lot of published information regarding the Tandy Color Computer (known as COCO) applies to the Dragon, and if you can get hold of a ROM disassembly for COCO it will give you most of the addresses that you will need in your Dragon.

Appendix H

A Useful Disassembler

A complete disassembler in machine code for the Dragon is not an easy project, but this BASIC program is useful for a large amount of the work that you can do on the Dragon ROM. It is based on the program by Brian Cadge that appeared originally in *Your Computer*, May 1983, and I am grateful to Mr Cadge and to the Editor of *Your Computer* for permission to reproduce the program. I have made some alterations to the original, which disassembled between start and end addresses in denary. The amendments permit a starting address to be entered in hex, and disassembly will continue until the BREAK key is pressed. A few lines of disassembly are shown on the screen, and remain until any key other than BREAK is pressed. This allows details to be copied if you have no printer. If you are using a printer, you might like to remove this feature when the printer option is taken. At each selection step, the ENTER key must be used.

Remember that a comparatively straightforward program like this cannot be expected to provide labels, nor to distinguish between instructions and tables of data. The word ERR will appear when the disassembler comes across a code that is not a valid instruction. There are also some differences between the output of this program, and the standards laid down by Motorola for 6809 assembly language. These are:

1. Immediate addressing omits the # symbol.
2. Both DP and extended addresses are shown within round brackets.
3. Indexed addressing, direct or indirect, uses square brackets.

Note that the ROM instructions start at $82F7, but there are ASCII coded words and a few tables of numbers embedded in the

ROM at higher addresses. When you enter this program, pay particular attention to the DATA lines. Each space and comma must be entered as shown – do not be tempted to change any of them.

```
10 '6809 DISASSEMBLER BY CADGE
20 '*****************************
30 CLS:POKE 155,80:POKE 154,64:P
OKE 153,16:POKE 328,0:'SET UP PI
NTER
40 CLEAR 5000:DIM A$(255,2)
50 FOR I=0 TO 255:READ A$(I,0),A
$(I,1),A$(I,2):NEXT
60 PRINT"6809 DISASSEMBLER FOR D
RAGON 32"
70 PRINT:PRINT:PRINT:SOUND 100,1
:SOUND150,1:SOUND200,1
80 PS=PEEK(65314)AND1
90 INPUT"PRINTER (P) OR SCREEN (
S)";OP$:IF OP$<>"P" AND OP$<>"S"
 THEN 90 ELSE IF
 OP$="P"THEN PR=-2 ELSE PR=0
100 IF PR=-2 AND PS=1 THEN PRINT
"ATTEND TO PRINTER!!":EXEC 41194
:GOTO80
110 INPUT"START ADDRESS IN HEX";
AD$:AD=VAL("&H"+AD$):ST=AD
120 IF AD<0 OR AD>65535 THEN PRI
NT" INVALID ADDRESS - REDO":GOTO7
0
130 CLS:PRINT@500:PRINT#PR,""
140 FOR I=AD TO AD+14
150 LI=I
160 SH=0
170 V=PEEK(I)
180 IF V=16 THEN SH=1:I=I+1:GOTO
170
190 IF V=17 THEN SH=2:I=I+1:GOTO
170
200 Z$=A$(V,SH):X$=RIGHT$(Z$,1)
210 IF X$<>"^" AND X$<>"#" AND X
$<>"$" AND X$<>"%" AND X$<>"&" A
ND X$<>"<" AND X
$<>">" AND X$<>"/" THEN P$=Z$ EL
SE P$=LEFT$(Z$,LEN(Z$)-1)
```

```
220 IF PR=0 THEN PRINTHEX$(I);TA
B(10);P$; ELSE PRINT#PR,HEX$(I),
P$;
230 IF X$="#" THEN 390
240 IF X$="%" THEN 410
250 IF X$="$" THEN 430
260 IF X$="&" THEN 460
270 IF X$=">" THEN 490
280 IF X$="<" THEN 720
290 IF X$="^" THEN 790
300 IF X$="/" THEN 820
310 IF PR=0 THEN 350
320 PRINT#PR,TAB(36);:FOR JJ=LI
TO I:HE$=HEX$(PEEK(JJ)):IF LEN(H
E$)<2 THEN HE$="
0"+HE$
330 PRINT#PR,HE$
340 NEXT
350 PRINT#PR,"":NEXT
360 K$=INKEY$:IF K$=""THEN 360
370 IF ASC(K$)<>3THEN AD=I:GOTO1
20
380 PRINT#PR," ":END
390 I=I+1:V=PEEK(I):PRINT#PR,HEX
$(V);
400 GOTO310
410 I=I+1:V=PEEK(I)*256+PEEK(I+1
):I=I+1:PRINT#PR,HEX$(V);
420 GOTO310
430 I=I+1:V=PEEK(I)
440 PRINT#PR,"("HEX$(V)")";
450 GOTO310
460 I=I+1:V=PEEK(I)*256+PEEK(I+1
):I=I+1
470 PRINT#PR,"("HEX$(V)")"
480 GOTO310
490 I=I+1
500 V=PEEK(I)
510 PRINT#PR,"I";
520 V=V AND 96
530 IF V=0 THEN P$="X"
540 IF V=32 THEN P$="Y"
550 IF V=64 THEN P$="U"
560 IF V=96 THEN P$="S"
570 PRINT#PR,P$;" ";
```

```
580 IF PEEK(I)>127 THEN 600 ELSE
 N=PEEK(I) AND 31
590  PRINT#PR,HEX$(N)"]";:GOTO71
0
600 V=PEEK(I)AND 159
610 IF V= 132 THEN PRINT#PR,"]";
:GOTO710
620 IF V= 136 THEN PRINT#PR,"+";
:GOTO390
630 IF V=137 THEN PRINT#PR,"+";:
GOTO410
640 IF V=134 THEN PRINT#PR,"+A]"
;:GOTO710
650 IF V=133 THEN PRINT#PR,"+B]"
;:GOTO710
660 IF V=139 THEN PRINT#PR,"+D]"
;:GOTO710
670 IF V=128 THEN PRINT#PR," INC
 1]";
680 IF V=129 THEN PRINT#PR," INC
 2]";
690 IF V=130 THEN PRINT#PR," DEC
 1]";
700 IF V=131 THEN PRINT#PR," DEC
 2]";
710 GOTO 310
720 I=I+1:V=PEEK(I)
730 P$=""
740 FOR J=7 TO 0 STEP -1
750 IF V<INT(2^J) THEN P$=P$+"0"
 ELSE V=V-INT(2^J):P$=P$+"1"
760 NEXT
770 PRINT#PR,P$" BIN";
780 GOTO310
790 I=I+1:V=PEEK(I)
800 PRINT#PR,HEX$(V);
810 GOTO310
820 I=I+1:V=PEEK(I)*256+PEEK(I+1
):I=I+1
830 PRINT#PR,HEX$(V);
840 GOTO310
```

```
850 DATA NEG $,,,ERR,,,ERR,,,COM
 $,,,LSR $,,,ERR,,,ROR $,,,ASR $
,,,ASL $,,,ROL $
,,,DEC $,,,ERR,,,INC $,,,TST $,,
,JMP $,,,CLR $,,,SFT,,,SFT2,,,NO
P,,,SYNC,,,ERR,,
,ERR,,,LIBRA %,,,LBSR %,,
860 DATA ERR,,,DAA,,,OR CC #,,,E
RR,,,AND CC #,,,SEX,,,EXG <,,,TF
R <,,,BRA ^,LBRA
/,,BRN ^,LBRN /,,BHI ^,LBHI /,,B
LS ^,LBLS /,,BHS ^,LBHS /,,BLO ^
,LBLO /,,BNE ^,L
BNE /,,BEQ ^,LBEQ /,,BVC ^,LBVC
/,,BVS ^,LBVS /,,BPL ^,LBPL /,,B
MI ^,LBMI /,,BGE
 ^,LBGE /,
870 DATA BLT ^,LBLT /,,BGT ^,LBG
T /,,BLE ^,LBLE /,,LEA X >,,,LEA
 Y >,,,LEA S >,,
,LEA U >,,,PSHS <,,,PULS <,,,PSH
U <,,,PULU <,,,ERR,,,RTS,,,ABX,,
,RTI,,,CWAI #,,,
MUL,,,ERR,,,SWI 1,SWI 2,SWI 3,NE
G A,,,ERR,,,ERR,,,COM A,,
880 DATA LSR A,,,ERR,,,ROR A,,,A
SR A,,,ASL A,,,ROL A,,,DEC A,,,E
RR,,,INC A,,,TST
 A,,,ERR,,,CLR A,,,NEG B,,,ERR,,
,ERR,,,COM B,,,LSR B,,,ERR,,,ROR
 B,,,ASR B,,,ASL
 B,,,ROL B,,,DEC B,,,ERR,,,INC B
,,,TST B,,,ERR,,,CLR B,,,NEG >,,
,ERR,,,ERR,,,COM
 >,,
890 DATA LSR >,,,ERR,,,ROR >,,,A
SR >,,,ASL >,,,ROL >,,,DEC >,,,E
RR,,,INC >,,,TST
 >,,,JMP >,,,CLR >,,,NEG &,,,ERR
,,,ERR,,,COM &,,,LSR &,,,ERR,,,R
OR &,,,ASR &,,,A
SL &,,,ROL &,,,DEC &,,,ERR,,,INC
 &,,,TST &,,,JMP &,,,CLR &,,,SUB
 A #,,,CMP A #,,
,SBC A #,,
```

```
900 DATA SUB D %,CMP D %,CMP U %
,AND A #,,,BIT A #,,,LDA #,,,ERR
,,,EOR A #,,,ADC
 A #,,,OR A #,,,ADD A #,,,CMP X
%,CMP Y %,CMP S %,BSR ^,,,LDX %,
LDY %,,ERR,,,SUB
 A $,,,CMP A $,,,SBC A $,,,SUB D
 $,CMP D $,CMP U $,AND A $,,,BIT
 A $,,,LDA $,,,S
TA $,,
910 DATA EOR A $,,,ADC A $,,,OR
A $,,,ADD A $,,,CMP X $,CMP Y $,
CMP S $,JSR $,,,
LDX $,LDY $,,STX $,STY $,,SUB A
>,,,CMP A >,,,SBC A >,,,SUB D >,
CMP D >,CMP U >,
AND A >,,,BIT A >,,,LDA A >,,,ST
A A >,,,EOR A >,,,ADC A >,,,OR A
 >,,,ADD A >,,
920 DATA CMP X >,CMP Y >,CMP S >
,JSR >,,,LDX >,LDY >,,STX >,STY
>,,SUB A &,,,CMP
 A &,,,SBC A &,,,SUB D &,CMP D &
,CMP U &,AND A &,,,BIT A &,,,LDA
 &,,,STA &,,,EOR
 A &,,,ADC A &,,,OR A &,,,ADD A
&,,,CMP X &,CMP Y &,CMP S &,JSR
&,,,LDX &,LDY &,
930 DATA STX &,STY &,,SUB B #,,,
CMP B #,,,SBC B #,,,ADD D %,,,AN
D B #,,,BIT B #,
,,LDB #,,,ERR,,,EOR B #,,,ADC B
#,,,OR B #,,,ADD B #,,LDD %,,,ER
R,,,LDU %,LDS %,
,ERR,,,SUB B $,,,CMP B $,,,SBC B
 $,,,ADD D $,,,AND B $,,,BIT B $
,,
```

```
940 DATA LDB $,,,STB $,,,EOR B $
,,,ADC B $,,,OR B $,,,ADD B $,,,
LDD $,,,STD $,,,
LDU 4,LDS $,,STU $,STS $,,SUB B
>,,,CMP B >,,,SBC B >,,,ADD D >,
,,AND B >,,,BIT
B >,,,LDB >,,,STB >,,,EOR B >,,,
ADC B >,,,OR B >,,,ADD B >,,,LDD
 >,,,STD >,,,LDU
 >,LDS >,
950 DATA STU >,STS >,,SUB B &,,,
CMP B &,,,SBC B &,,,ADD D &,,,AN
D B &,,,BIT B &,
,,LDB &,,,STB &,,,EOR B &,,,ADC
B &,,,OR B &,,,ADD B &,,,LDD &,,
,STD &,,,LDU &,L
DS &,,STU &,STS &,,,,,,
```

Index

MORE SPEED, MORE POWER AND GREATER CONTROL!

Sooner or later most users feel restricted by BASIC's limitations such as the slow speed of some commands – particularly when fast animation of graphics is concerned – and the limited control over the machine. For really fast operation and full mastery over the machine, making special effects such as new graphics modes possible, the best answer is machine code.

Machine code consists of number codes which affect the microprocessor of the computer directly. The use of machine code by-passes BASIC altogether, so that the instructions which you write and use in machine code will exert direct control over your Dragon. This book assumes nothing more than a reasonable knowledge of BASIC. You are shown what machine code is, how it works, and how to enter, run and save code. In the course of reading this book, you will learn much more about how the Dragon works. You will also discover how versatile you can become, as a whole new world of special effects opens up. The vital aids to efficient machine code writing are also covered, for you to go further with this fascinating subject. Many who do so never return to BASIC again!

The Author

Ian Sinclair is a well known contributor to journals such as *Personal Computing World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over forty books on electronics and computing aimed mainly at the beginner.

GRANADA PUBLISHING
Printed in Great Britain 0246122249

£7.95 net

SINCLAR **INTRODUCING DRAGON MACHINE CODE**