# DRAGON 32

# MACHINE CODE

FOR BEGINNERS

COMPUDAT

# DRAGON 32 MACHINE CODE FOR BEGINNERS

ACKNOWLEDGEMENTS

Many thanks go to Dave and Ray of LEICESTER MICRO CENTRE for lending me the word processor to write this book; to Chris for practical help and to Kay and Paul for keeping out of my way!

# CONTENTS

# PREFACE

After the first flush of excitement on opening my brand new Dragon 32 computer was over and I'd programmed many a horrendously slow Space Invaders Game in Basic I began to hanker after the real thing. Machine Code Programming!! That's for me I thought. I'll just pop out and get a book about it. That's where the trouble started.

The precious few books that I could find all assumed you had a maths degree, an IQ of at least 205 and an intimate knowledge of such strange phrases as Extended Indirect Addressing, Non Maskable Inputs and, worst of all, FIRQ and Nibble.

There was nothing that started right at the beginning and spoke in simple terms. This may be all right for the new breed of computer whizz kid but for someone as long in the tooth as me something else was needed.

It is with this sort of problem in mind that I have written this book. I hope it makes your entry into the fascinating world of Machine Code a little less tortuous than mine.

I'm assuming that you have access to a Dragon 32 computer, understand Basic programming reasonably well and have infinite patience.

I always find I can understand something a lot easier if I am shown a practical example so this is the system I've adopted in this book. We will learn Machine Code by doing lots of small practical programmes step by step and later on I'll show you how to put these together to form quite complex but easily understood programmes.

In order to make things easier to understand I've had to over simplify the amazing workings of the Dragon. I shall not apologise to the purists, who this may upset, because I believe they cause all the confusion in the first place.

Don't forget, you don't need to know anything about Machine Code to use this book. I have assumed you have no previous knowledge at all. When you have finished this book you will not be an expert programmer but you will have a good basic understanding of Machine Code, be able to write useful programmes and be ready to progress to more advanced work.

Always remember, especially with Machine Code programmes; "Banging and swearing at a computer never helps!! But it does make you feel better."

# WHY MACHINE CODE

What is the use of Machine Code and why is it better than Basic programming? Well in many cases It's not better than Basic. It's complex, takes a long time to write and It's a lot more difficult to debug if you make a mistake. But having said that, programmes writen in Machine Code(M/C) run a lot faster than Basic programmes so if you want to write fast action arcade type games you need to use Machine Code. Also there are lots of tricks you can do using M/C that aren't possible in Basic. Let's try a few:-

Switch on your Dragon and type the following:-

PRINT MEM        and press the ENTER key.

24871        will appear on the screen. That's the amount of free memory you can use. Not 32000 as you might have thought because a large amount of memory in the computer is reserved for High Resolution Graphics which I found a bit disappointing when I found out; but never fear, we can steal that extra memory back. Type the following:-

POKE 25,6        and press the ENTER key. Nothing happens.

Now type NEW and press the ENTER key and again nothing happens but now if we type:-

PRINT MEM        and press the enter key

31015 appears on the screen. 6k of extra memory for you to use and that's near enough to 32k for me. Don't worry about how it works I'll explain that later.

Don't worry about all this Poking, it

definitely can't damage your computer in any way. Poke, by the way, tells the computer to store something in a certain part of it's memory. More explanations later.

Now try this. Plug in your tape recorder to your Dragon but don't put in a tape. Press the rewind button instead. Now type the following:-

POKE 65313,60 and press the ENTER key. Your tape motor will burst into life. This M/C instruction is the same as the Basic MOTORON command. Now type:-

POKE 65313,52 and press the ENTER key and the motor will stop. Now try typing:-

POKE 65495,0 but before you press the enter key look at the flashing cursor and see how fast it is flashing. As soon as you press the ENTER key it begins to flash much quicker.** You will also find basic programmes will run much faster but you will not be able to LOAD or SAVE any programmes until you press the reset button. On some computers this does not work. Don't worry if it dosen't, just switch off and on again.

Clever isn't it. Now switch off your machine and read on, there are more secrets to be revealed!!

**There is some controversy about whether this command can damage your machine. All I can say is I use it a lot and my Dragon's fine.

## WHAT'S A MEMORY MAP

At this point in time we need to look inside your Dragon. Don't panic, you won't need a screwdriver. We're going to examine your Dragon's memory.

The mass of electronic wizardry inside your computer can be divided into the following blocks:-

1 The Central Processing Unit or CPU that does all the clever work and calculating and is really the heart of the computer. In the Dragon 32 it's a silicon chip called a 6809E and is one of the most advanced 8 Bit microprocessors in the world. It's like a brain and like a brain it needs to communicate with the outside world. To do this it needs:-

2 Input Output Devices or PIAs which are again silicon chips which the Central Processing Unit(CPU) uses as it's eyes, ears and mouth. They allow you to tell the CPU what you want to do via the key board, tape or joystick and the CPU to tell you what it's done via the TV screen. But a brain is of no use if it can't remember so it needs:-

3 Memory. This is one of the most confusing parts of the computer so we will look at it in some detail. There's no need to worry it's quite easy if we look at it step by step. That's what the computer does.

The memory of the computer is where we store all the information the computer needs and as it needs many thousands of pieces of information we must be very careful where we store them or they might get lost. If we think of the computers memory as a very long street with houses on it then in each house we can store one small piece of information. We can remember where we stored this information because each house will have a number; it's address or as we shall call it it's "MEMORY ADDRESS". The Dragon has a very long street. We'll call it Memory Lane (sorry about that). It has 65,536 separate houses or Memory Addresses (64k).

In the earlier example when we typed:-

POKE 25,6 what we were in fact doing was telling the Central Processing Unit (CPU) to go to Memory Address number 25 and store in it the number 6. This it did and, as we saw, it released more memory space for us to use. More about this later. First we must look at this memory in more detail.

I have said that the Dragon has over 65,000 memory addresses but you know that we can only use 24,000 or 24k of these. What's happened to the other 40,000? To explain this let's look at Fig 1. This is a map of our Dragons street of memory addresses. The addresses start with 0 at the top and go all the way down to 65,535 at the bottom. (To a computer, 0 is a number, so counting always starts at 0 not 1.) We will call it by it's proper name:- "A MEMORY MAP".

The chart in Fig 1 shows the following:-

<u>1</u> The Memory Addresses. I have put these in blocks because for the time being we are only interested in how the computer uses sections of its memory. Looking at what each individual address does comes later.

<u>2</u> The Hexadecimal Address. Hexadecimal or Hex numbers are just a different way of expressing a decimal number. We will need to learn these later but for the time being we will

| Memory Address |  |  | What it does | Mem | Type |
|---|---|---|---|---|---|
| Dec | Hex |  |  |  |  |
| 0 – 1023 | 0000 – 03FF | A | System Use | 1k | Ram |
| 1024 – 1535 | 0400 – 05FF | B | Used by computer to store what is to be displayed on screen (normal) | ½k | Ram |
| 1536 – 7679 | 0600 – 1DFF | C | Used to store High Res. screen display | 6k | Ram |
| 7680 – 32767 | 1E00 – 7FFF | D | Used by you to store your programs and variables | 24k | Ram |
| 32768 – 49151 | 8000 – BFFF | E | Basic Interpreter Used by computer to convert Basic | 16k | Rom |
| 49152 – 65279 | C000 – DFFF | F | Reserved for Cartridge or expansion | 16k |  |
| 65280 – 65535 | FF00 – FFFF | G | System Use | ½k |  |

FIG 1

ignore them and use the more easily understood decimal numbers.

3 What it does. This shows roughly what use the computer makes of each block of memory addresses or locations. We will be looking at each block in detail.

4 Approximate amount of memory. This shows approximately the size of each block of memory addresses or locations and is expressed in "k" or thousands of memory addresses. 1k means 1 thousand memory locations in computer jargon. Unfortunately it's not that simple. For reasons, that I hope will become obvious, 1,000 to a computer is infact 1,024. I have said that the Dragon has 64k of memory addresses. If you multiply 1024 by 64 you will get the magic figure 65,536.

5 Type of Memory. I keep saying that the Dragon has 64k of Memory Addresses. This is not the same as saying the Dragon has 64k of Memory. What it means is that the Dragon has the ability to use or address 64k of memory. What the designers of the Dragon have done is to split this 64k of memory addresses into blocks that the Central Processor (CPU) uses in different ways.

There are 2 main types of Memory that we are concerned with:-

A Random Access Memory or RAM. This type of memory consists of silicon chips that can store information. We need not concern ourselves with the electronics of how it works just as long as we imagine it as part of our street of addresses. These addresses we can use or access to store information. We can put information in or take it out in a random manner. We can also alter the information in each address.

<u>B</u>   Read Only Memory or ROM. This again consists of silicon chips that form part of our street of addresses only this time we can only Read the information in each address. We cannot put information in or alter it. The information in the ROM silicon chips is physically burnt in during manufacture. This means that if the Dragon is switched of the information contained in ROM is not lost whereas anything contained in RAM will be gone forever.

If we look at section "A" of the chart we see it uses memory addresses 0 to 1023 (The first memory location is called 0 not 1 and the last is 65,535. This gives a total of 65,536). It is allocated, by the designers of the Dragon for, System Use. What this means is that the CPU needs a certain amount of memory for it's own use. In here it stores information about what the computer is doing. Whether the printer is being used, where the basic programmes are stored and a host of other information. Normally, when we use Basic, we are not aware of this section because the CPU looks after it but if we use Machine Code we can access this section and store information in it that will make the CPU do what we want it to do not what it wants. We have already done this. The CPU normally only gives us 24k of memory addresses to store our programmes in. By storing the number 6 in memory address 25 (which is part of section "A") we forced the computer to give us more memory for storing programmes.

Section "B" is used by the CPU to store information that will be displayed on the screen in the Low Resolution or Text Mode. To do this it uses a system similar to the PRINT @ GRID in the back of the Dragon's Basic Manual. Here the screen is divided into boxes numbered 0 to 511. The CPU thinks that memory location 1024 is equal to box number 0, memory location 1025 is equal to box number 1 etc. through to memory ocation 1535 being equal to box number 511. gain the CPU normally controls this section but e can use it with Machine Code. Open your ragon Basic Manual at Appendix A.

This shows the ASCII codes for all the characters that can appear on the screen. ASCII code numbers are a standard set of numbers allocated to the characters that the computer uses and are <u>supposed</u> to be the same the world over.

If we type, on the Dragon, POKE 1230,65 and press the ENTER key the letter A will appear in the centre of the screen. What we have done is to store the ASCII code for A (65) in memory location 1230 and the CPU has automatically assumed we want to display it on the screen. We can use this to place any character at any point on the screen. Experiment by POKEING different numbers into memory address 1230. Just change the number after the comma to the ASCII code number you want to appear.

Using this method we can put a character in the bottom right hand corner of the screen without the screen automatically scrolling up a line. Just type POKE 1535,65 and see what happens.

This ability to make characters appear on the screen is one which we will use a lot when we begin to program in Machine Code.

Section "C" is very similar to section B because the CPU uses this to store information that it wants displayed on the High Resolution screen. If we type POKE 2000,65 we can store this information in the High Resolution area of memory. It will not appear on the screen because we have not asked the CPU to go to the High Resolution Display Mode yet.

The size of this area can be increased as we use more pages of graphics(see page 92 of the Dragon Manual) and when it does it borrows memory addresses from section "D".

Section "D" is the area that we use the most. In it the CPU stores all the Basic programmes and Variables that we type in at the keyboard. We will also be using it to store our Machine Code programmes but as we shall see it is not the only place that we can use.

Section "E" is very interesting. In it is stored the Basic Interpreter. The CPU, which as we have said does all the clever work in our computer, can only understand Machine Code. The Basic interpreter is a set of instuction, writen in Machine Code, that converts the Basic commands that we type in on the keyboard into Machine Code that the CPU can understand. The instuction are like a set of subroutines that convert each Basic word into M/C.

These instructions are in Read Only Memory or ROM and as they are burnt in during manufacture we cannot alter them. That does not mean, however, that we cannot use some of these instructions in our M/C programmes.

Section "F" caused me the most confusion when I first became interested in M/C. Cartridge Memory it said in my book but I couldn't work out why I couldn't store anything in these memory addresses.

I said before that the Dragon has 64k of Memory Addresses not 64k of Memory. This section shows what I meant. This area is like part of a street where no houses have been built. The plans are all drawn and addresses allocated to the houses but until we build the houses we have nowhere to store our information.

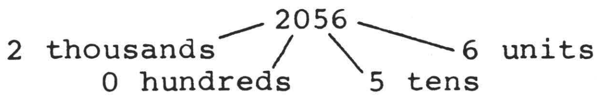
When we plug a cartridge into the Dragon we add these houses. The memory we add, however, is ROM so the computer can only read the instructions stored there and carry them out. We could, however, plug RAM memory into the Dragon expansion port and then we could store information there. This part of the memory map is very useful if we want to add other devices to the Dragon.

Last of all we have section "G" which like section A is reserved for System Use. But once again we can access it to make the computer do what we want.

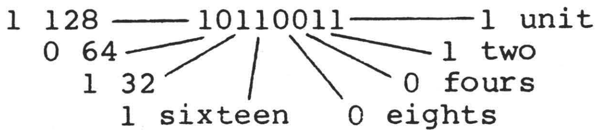
That is a very brief look at the Memory Map of the Dragon 32 computer. We will look at it in more detail as we start to programme but first we need to learn what information we can store in these memory addresses and how it affects the computer.

## BITS, BYTES, POKES and PEEKS

The electronics of the Dragon can only understand two things. On and off. So, to talk to it we use Binary Numbers. Don't panic, we're not going to dwell on this too long but we do need to know about sets of Binary numbers. Lets look at a decimal number and how it is made up:-



A Binary number does not count in 10s, as we do, but in 2s. When we count in Decimal, every time we count to 10 we carry 1 into the next column and so on. In Binary, we count to 2. This makes Binary numbers look very long and cumbersome but computers find them easy. Let's look at a Binary number:-



If we add all these together we find that the Binary number 10110011 is equal to the decimal number 179 (128+32+16+2+1). All we really need to understand about this is that the computer can represent this by saying that 0 = off and 1 = on. It can then use a simple set of switches to represent a number. This may appear difficult to us but a computer can alter it's switches so quickly it doesn't matter.

The Binary number above consists of 8 figures. Each one is called a Bit and 8 Bits together (as the number above) is called a BYTE. 4 Bits or half a Byte is called, would you believe it, a NYBBLE. If we add it up the highest number we can write in binary using 8 its or 1 Byte is 11111111 which is equal to the ecimal number 255.

Now the 6809E CPU used in the Dragon is an 8 Bit device. That means that it handles it's Binary numbers 8 Bits or 1 Byte at a time. Remember this,we will be using it later.

Binary numbers are difficult to write and as the CPU needs 8 Bits at a time it may be easier to use number systems that count in 8s. These are known as OCTAL numbers. In fact, it is easier to use a system that counts in 16s and is known as a HEXIDECIMAL number.

If our CPU counts 8 Bits at a time it can count up to 255. That's the same number as there are ASCII codes and if our CPU counts 16 Bits at a time the highest number it can count is 65,535. This is the number of memory addresses the Dragon has. This is not a coincidence. The Dragon cannot easily have more memory addresses than 65,535 because it can't count any more.

Here are a few Hexadecimal or HEX numbers with decimal equivalents which may make it a little clearer.

| HEX | | DEC | HEX | | DEC |
|---|---|---|---|---|---|
| 1 | = | 1 | A | = | 10 |
| 2 | = | 2 | B | = | 11 |
| 3 | = | 3 | C | = | 12 |
| 4 | = | 4 | D | = | 13 |
| 5 | = | 5 | E | = | 14 |
| 6 | = | 6 | F | = | 15 |
| 7 | = | 7 | 11 | = | 16 |
| 8 | = | 8 | 12 | = | 17 |
| 9 | = | 9 | 13 | = | 18 |

As you can see this system uses both etters and numbers and although it seems a ittle odd it does have great advantages when dealing with computers.

All 8 Bit numbers can be represented by 2 figures ie. FF in Hex = 255 in Decimal. 16 bit numbers can be written using 4 figures ie. FEBD in Hex = 65213 in Decimal. Once you get used to this Hex numbers are very useful but for the time being we will stick to Decimal numbers. When I do use Hexadecimal numbers I will put &H in front of them. The Dragon understands this, so should you.

You can convert Decimal to Hex numbers using the Dragons HEX$ command (see page 70 of the Dragon manual). You can also convert Hex numbers to decimal using the following programme.

```
10 CLS
20 INPUT X$
30 H$ = "&H" + X$
40 PRINT VAL ( H$ )
```

Well that's Bits and Bytes, what about Peeks and Pokes. If you tell your Dragon to POKE 65312,134 it will look for memory address 65312 and store in it the number 134. It will automatically convert these numbers to Binary so that the CPU can understand them (clever thing). Don't forget you can't Poke anything into ROM. You won't harm it, it just ignores it.

If you type PRINT PEEK (65312) the Dragon will go to memory address 65312, look in and display on the screen what's stored there. In this case 134 because we've just put it there.

You cannot store numbers greater than 255 in any one memory address (try it and see but don't say I didn't tell you so). This is because the CPU is an 8 Bit device and, as we said earlier, cannot handle numbers greater than 255. If you type POKE 65312,257 you will get an error.

Now, we've done all the boring but necessary ground work, we can look at Machine Code Instructions.

# THE INSTRUCTION SET

I have said that the 6809 CPU is the clever bit of the Dragon that does all the calculating and controls how the computer works. But how does it do this and how does it know what to do? The answer to this is simple. You tell it what to do by giving it a set of instructions to follow. The 6809E microprocessor (CPU) has 59 instructions that it can obey and by using them in different ways this can be increased to over 1,400. You can store a list of these instructions in the computers memory and what the CPU will do is to go to a memory address, look into it, read the instruction stored there and carry it out. There are many instructions it can perform. For instance, the instruction in the memory might tell it to go to another memory address, in the part of the memory map reserved for screen display (section B Fig 1) and there store the ASCII code for "A". This would cause the letter "A" to be displayed on the screen.

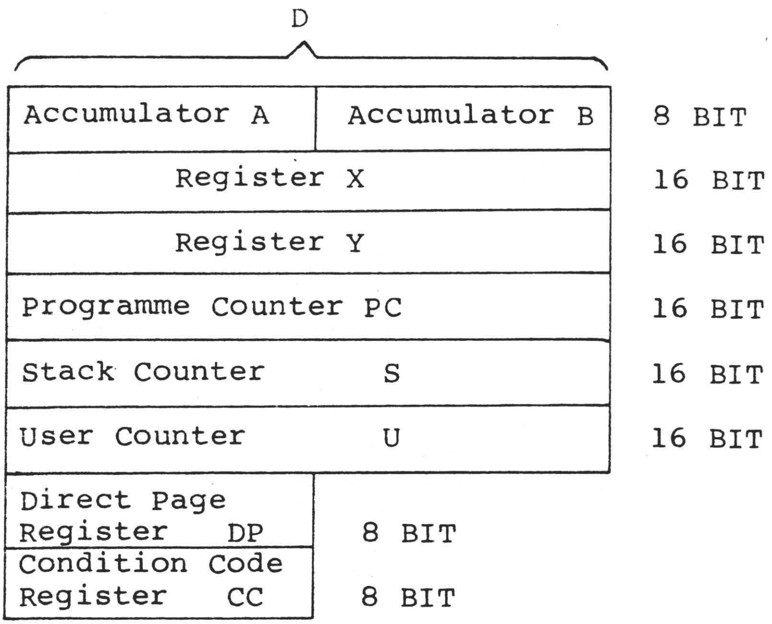
Before we look at the Instruction Set in detail we need to look inside the CPU and see what it contains and how it works.

I should point out that different types of microprocessor have different internal structures and the instruction set for the 6809E would be different to, for instance, the Z80A used in the Sinclair Spectrum. The two sets of instructions cannot be interchanged. Therefore M/C programmes for the Dragon will not work on a Spectrum or any other make of machine and vice versa. We can, however, once we understand how the machine works, get programmes written for a 6809E CPU to work on another machine using the same CPU (a Tandy Colour Computer for instance) but we will have to alter the programme to take account of the the different memory map layout nd controls of the other machine.

Fig 2 shows a block diagram of the nside of the 6809E microprocessor.

FIG 2

6809E INTERNAL STRUCTURE



We can see that the CPU contains several oxes called Accumulators and Registers. The ay they work is a little bit like the memory ddresses contained in the memory map. They act ike stores that the CPU can put information in, ake it out and move it all about (just like oing the Hokey Cokey). Also the CPU can act on ts own internal Registers and Accumulators and lter the contents in line with the instructions hat we give it. Let's look at each part in etail.

ACCUMULATORS A and B are, as far as we re concerned at the moment, identical. Each one can hold an 8 Bit or 1 Byte number. We can put numbers into these and manipulate them using the CPU's instruction set. As each accumulator can only hold an 8 Bit or 1 Byte number this means that we can only work on 1 Byte at a time. If we want to work on a longer number we must split it up into sections 1 Byte long.(This is easy to do if we use Hexidecimal numbers but not so easy in Decimal)

The 6809E is rather clever in that it can combine accumulators A and B (calling them Accumulator D) and for certain operations treat it like a 16 Bit or 2 Byte number. This gives it considerable power over an 8 Bit micro. The way it combines these accumulators is to store the first Byte of the number in accumulator A and call this the Most Significant Byte (MSB) and the second part or Byte is placed in accumulator B and is called the Least Significant Byte (LSB). ie:-

```
10110110:10101101   16 Bit number
  MSB   :  LSB
```

The X and Y REGISTERS again can be considered, for the moment, as being the same. They can store 16 Bit numbers and are usually used to store memory addresses associated with the numbers in the accumulators. For instance, we can tell the CPU to go to a memory address stored in register X and store in it the number in accumulator A.

This may seem a little long winded but as we shall see later on we can manipulate these internal registers to give us very powerful command facilities. Also we can tell the CPU to go to the memory address stored in the Y register and read the instruction stored there. This instruction may tell it to go to another memory address, get the number stored there and act on it in some way, maybe add to it or subtract and then store the result in another memory address. The CPU performs most of its tasks like this.

This may appear to be a tedious way of performing, when, using Basic, you just type PRINT 2+3, and believe me it is tedious. Don't say I didn't warn you that M/C programmes are very long to write. And as with Basic programming we have to be very careful to write the instructions in a clear and logical manner so with M/C programmes we have to be doubly careful.

The PROGRAMME COUNTER (PC) is used by the CPU to remind itself where in the programme it is. It contains a 16 Bit number that usually points to the next memory address of the next instruction the CPU will use.

The STACK COUNTER (S) can again be used by the CPU to keep track of itself and is usually used when the CPU jumps to a subroutine to tell it where to return.

The USER STACK (U) can be used by you, the programmer, to store memory addresses you may want to use.

The DIRECT PAGE REGISTER(DP).The computers memory map can be thought of as a book with 256 pages each containing 256 memory addresses listed in numerical order, hence address 200 would be on page 1 and address 270 on page 2. The DP register keeps a track of which page of addresses the CPU is using. Its use will become clearer as we start programming.

The CONDITION CODE REGISTER(CC). This register keeps a record of the condition of the CPU and we will learn more about it later.

But now let's write a programme at last.

# A PROGRAMME AT LAST

Let's look at one of the CPU's 59 instructions:-

LDA

Great, at long last a machine code instruction. But what does it mean? Actually it's quite simple when you know. It's a mnemonic, which is easier to understand than say,or memory jogger for the instruction " Load Accumulator A" with a number.

Now how do we get this instruction into the CPU? Again it's simple when you know how. The CPU can only understand numbers, so first we must represent LDA by a number. This has already been done by the CPU manufacturers and in appendix A are listed several instructions with the numbers that represent them. In this case the number representing LDA is 134. Now we need to get this number into the CPU. In fact it's easier to get the CPU to come to it. To do this we will store the number in a memory address by typing POKE 2000,134. This will store the number 134 at memory address 2000. If we now type EXEC 2000 the CPU will go to memory address 2000, read the instruction and carry it out.

But now we have a problem. We have told the CPU to load it's accumulator A but with what? Well in fact we have told it what. There are several different ways of using the LDA instruction. We have used the one called " Immediate Addressing." This means that the CPU will automatically assume that the number it is to load in accumulator A is stored in the next memory address. That is address number 2001. When we type EXEC 2000 the Programme Counter (PC) register will contain 2000. The CPU will go to this address and read the instruction (134). This will tell it to load accumulator A.The CPU will then go to address 2001 and store this number in it's A accumulator.

How does the CPU tell the difference between instruction numbers and ordinary numbers? It doesn't. It relies on us to get things in the right order. If, for instance, in the last example we stored instruction number 107 in memory address 2001 the CPU would think it was an ordinary number because it's last instruction (134 stored at address 2000) told it it was. As you can see we must be very careful.

And now, I'm afraid, we must learn some boring conventional terms that the computer world has decided to adopt.

The instruction code is called the OPERATION CODE and is usually expressed as a Hexidecimal number. Thus the Operation Code for LDA is &H 86 (don't forget I said I'd put &H in front of all Hexidecimal numbers so that you recognise them).

The number to be acted on (stored at address 2001 last example) is called the OPERAND and the memory address that the CPU will go to to find it is called the EFFECTIVE ADDRESS.

That will do for now but I'm afraid there is more to come. But first let's look at some more instructions.

## MORE INSTRUCTIONS

We will now look at a few more commands and give a brief description of how each one works. Then we can write a short programme to show how they all fit together.

### STA

This instruction tells the CPU to store the contents of it's accumulator A in a memory address. Again we can ask the question, as we did with LDA, how does the CPU know what address to use?

When we talked about the instruction LDA we said we were using "Immediate Addressing" mode and this meant that the number we were to load into accumulator A would be in the next memory address after the address containing the instruction code. For the instruction STA let's use a different mode called "Extended Direct Addressing".(We will be looking at these different addressing modes in more detail in the next chapter so don't worry if you can't grasp it yet.)

The Operation Code (Instruction Code) for STA,in this addressing mode, is 183 or in Hexidecimal &H B7 (Don't forget the &H). The memory address will be found in the next memory address after the one containing the Operation Code. This is not quite true because as the memory address will be a 2 Byte or 16 Bit number it will need 2 memory addresses to store it.

Let me try and explain that a bit better. When we were looking at how the A and B accumulators combined to store a 2 Byte or 16 Bit number I said that the first part of the number(MSB) was stored in A and the second half(LSB) was stored in B. We do the same after the instruction STA. The first address the CPU looks at will contain the operation code 183 which will tell the CPU to store the contents of it's accumulator A in a memory address. The next address will contain the first half(MSB) of the memory address that the CPU is to store the contents of accumulator A in and the next address will contain the last half(LSB).

| | | | |
|---|---|---|---|
| 1st address | 3000 | contains 183 | operation code |
| 2nd address | 3001 | contains MSB | address to |
| 3rd address | 3002 | contains LSB | store A |

The instruction STA therefore requires 3 memory address to give the complete information the CPU needs.

## CLRA

This instruction erases the contents of accumulator A and leaves it empty. The Operation Code is 79 or in Hex &H 4F. As this instruction only affects the CPU itself and does not refer to another memory address, it only requires 1 memory address. This is known as "Inherent Addressing".

The last instruction we shall look at for the moment is:-

## ADDA

This adds a number to the number already stored in accumulator A. The Operation Code we will use is 187 or in Hex &H BB. The addressing mode we are using is "Extended Direct Addressing" again. This means that the CPU will have to look in the memory address given after the instruction to find the number it is to add to accumulator A.

Let's write a short programme using the instructions we have learned so far.

In this program we will add a number to a number we have already stored in a memory location, then place it in the section of memory reserved for graphics display so that the computer thinks it's an ASCII code and displays it on the screen. We will make the letter S appear (ASCII code 83).

This will be the order of instructions:-

1 Store number 43 in memory address 25000
2 Clear accumulator A
3 Load accumulator A with number 40 which is stored in the next memory address(20002).
4 Add the number stored in memory address 25000 to the number already in accumulator A.
5 Store the new contents of accumulator A in memory location 1200 (part of the graphics display area).

This is how we do it. Type:-

```
POKE 25000,43    Stores 43 in location 25000

POKE 20000,79    Operation code to clear A
POKE 20001,134   Operation code for LDA
POKE 20002,40    Number to be loaded in A
POKE 20003,187   Operation code for ADDA
POKE 20004,97  } Address were number
POKE 20005,168}  is stored
POKE 20006,183   Operation code for STA
POKE 20007,4   } Address were result
POKE 20008,176}  is stored
POKE 20009,57    RTS returns control to basic
```

After you've typed this in press the CLEAR key to clear the screen. We can then tell the CPU to go to memory address 20000 and carry out the instruction it finds there. To do this just type EXEC 20000 and press the ENTER key. Low and Behold, a letter S appears in the middle of the screen which, amazingly, is what's supposed to happen. The OK prompt will appear back on the screen meaning that the Dragon has completed your Machine Code routine and returned control of the computer to the Basic Interpreter. This was done by the RTS instruction. The Dragon has treated the M/C routine as a it would a Basic subroutine but if we did not put the RTS instruction at the end of our program the Dragon would not know what to do after it had run the M/C and would have crashed.

You may also be confused by locations 2004,2005 and 2007,2008. These are supposed to show 16 BIT (2 BYTE) memory addresses but in fact they show 97,168 and 4,176. If, however, we convert these numbers to HEX they become:-

&H 61,A8 and &H 04,B0

Now go to your Dragon and type:-

PRINT &H 61A8 and press ENTER

25000 will appear

Try PRINT &H 04B0 and 1200 will appear

It is easier to express 16 BIT memory addresses as 2 one BYTE Hexadecimal numbers so start practicing converting Decimal numbers to Hex and vice versa. We shall be going over this subject again later so don't worry to much if you haven't understood.

## ADDRESSING MODES

What are these addressing modes I keep going on about? They're very important that's what they are. If you can master addressing modes then you're 80% of the way to mastering machine code.

Addressing modes can best be described as, where the CPU goes to find or store the data it needs to perform a machine code instruction. Lets look at it step by step.

In the last example we used the M/C instruction LDA which we said tells the CPU to load it's A accumulator with a number. To find out where this number is the CPU also needs to know what addressing mode you want it to use. For instance:-

IMMEDIATE ADDRESSING tells the CPU that the number it needs to load into the A accumulator is stored in the memory address <u>immediately</u> following the memory address that contained the operation code (LDA instruction).

Example:-

```
        Memory address 25000 contains LDA
        Memory address 25001 contains number 43
```

The CPU will go to memory address 25000 were it will see that it has to load its A accumulator with the number 43. We say that the "EFFECTIVE ADDRESS" or the memory address were the data is found or stored is immediately after the operation code. Hence Immediate Addressing.

EXTENDED DIRECT ADDRESSING tells the CPU that the "Effective Address", or the memory address were the data it needs is stored, is pointed to by the next memory address. It sounds a little confusing dosen't it. Let's look at it gain:-

In Immediate Addressing the next memory ddress, after the operation code, actually ontains the data the CPU needs.

In Extended Direct Addressing the next emory address actually contains the location of nother memory address that the CPU must go to o find the number it needs.

xample;-

Memory address 20000 contains LDA

Memory address 20001 contains another emory address 15000

Memory address 15000 contains the number 6

The CPU will go to memory address 20000 ere it will see that it has to load its A ccumulator with the number stored in memory ocation 15000 (number 86).

As you can see this is a very useful ddressing mode because it allows the CPU to ove about all over the memory map collecting nformation where it likes. If we use the STA nstruction in this addressing mode, which we did in the last programme, we can write a machine code routine in part D of the memory map (see page 7) that will store data in part B of he memory map (the screen display area). This s one way that we can use machine code to isplay information on the screen.

You may have noticed I said that the memory address following the operation code contained the memory address that the CPU must go to in order to collect the data it needs. In fact a memory address will be a 16 BIT (2 BYTE) number and as each memory location can only hold an 8 BIT (1 BYTE) number we will have to use the next 2 memory locations after the operation code to contain the address the CPU must go to (the Effective Address).

Example:-

```
        Memory address 20000 contains LDA
        Memory address 20001 }
        Memory address 20002 } contain address
where the data is stored.
```

How does the CPU know which addressing mode it is to use? It's simple really. As we said before, each M/C instruction is represented by a number. We only use the mmenomic LDA as it's easier to remember.

In Immediate Addressing the number for LDA is 134

In Extended Direct Addressing the number for LDA is 182

Just give these numbers to your CPU and it works the rest out for it's self. Clever little thing.

Let's do a practical example to show the difference in the 2 different modes. We'll write a short program to write "HELLO" on the screen. How exciting!! To make it a little different we'll put it on the bottom line but without the screen automatically scrolling. Try doing it Basic first.

We need to use the Dragon manual Appendix A which shows the ASCII character codes. Use the "With Shift Key" column.

We are going to load the A accumulator with numbers that represent the ASCII characters then store the contents of the A accumulator in the area of memory used for the screen display. The LDA operation code will use Immediate Addressing and the STA operation code will use

Extended Direct Addressing.

This will be the order of instruction:-

1 LDA with 72 (ASCII H)
2 STA at memory address 1520(screen display)
3 LDA with 69 (ASCII E) 4 STA at memory address 1521(next address on screen)
5 LDA with 76 (ASCII L)
6 STA at memory address 1522(next address on screen)
7 STA at memory address 1523(note accumulator A already contains 76 so we don't need to load it again)
8 LDA with 79 (ASCII O)
9 STA at memory address 1524
10 RTS returns control to basic

Note that the LDA operation code wipes out what was previously in the A accumulator wereas STA leaves the accumulator intact (lines 6&7).

Now we can load this programme into the Dragons memory using the Poke command as before but to save us a lot of typing we will write a short program in basic to do it for us.

Enter the following Basic programme:-

```
10 CLS
20 INPUT"ENTER START ADDRESS";S
30 FOR P=S TO 32000
40 INPUT N
50 POKE P,N
60 NEXT P
```

This will ask us to enter the number of the memory address where we want the M/C programme to start. It then uses a FOR NEXT loop to poke code numbers into memory as we input them. You will have to use the BREAK key to stop the programme after entering the last number.

Run the above programme and when it asks for the start address enter 18000. You can start the programme any where you like between 10000 and 30000 (try it and see). That's because this programme is Position Independent, which means it will run anywhere in memory (within reason). Most M/C programmes are not. Then enter the following list of numbers (explanations are listed on the right).

| Number | Explanation |
|---|---|
| 134 | LDA Immediate Addressing |
| 72 | with 72 (ASCII H) |
| 183 | STA Extended Direct Addressing |
| 5 | 2 Bytes to give memory address 1520 |
| 240 | (next chapter explains fully) |
| 134 | LDA |
| 69 | with 69 |
| 183 | STA |
| 5 | memory address 1521 |
| 241 | |
| 134 | LDA |
| 76 | with 76 |
| 183 | STA |
| 5 | memory address 1522 |
| 242 | |
| 183 | STA |
| 5 | memory address 1523 |
| 243 | |
| 134 | LDA |
| 79 | with 79 |
| 183 | STA |
| 5 | memory address 1524 |
| 244 | |
| 57 | RTS returns control to Basic |

Press BREAK key

Now press the CLEAR key and type
EXEC18000
Or whatever start address you used

HELLO will appear on the bottom line.

Some things to notice are that HELLO will always appear on the bottom line no matter where the flashing cursor is. Also typing NEW will have no effect on your M/C program. It will sit stubbornly there in memory until you either switch off or write over the top of it by mistake. If you started at address 18000 then leave it there for the time being and we'll use it later.

You will no doubt have noticed how tedious it was entering a new memory location every time we used STA. Well, we can use another addressing mode, called AUTO INCREMENT, to help us a lot. Turn to the next chapter for more startling revelation!!

# SOME MORE TRICKS

I'm afraid you've been conned. You've come rushing to this chapter expecting to find out all about Auto Increment but first you have to learn a little bit more about Hexadecimal numbers!! In the last example we used 2 memory locations, after the STA operation code, to store memory address 1520 but we didn't put 1520 in these locations, we put 5,240. in order to explain why we must first convert the decimal number 1520 to Hexadecimal and we will use the dragon to do it:-

Type:- PRINT HEX$(1520) and press ENTER

5F0 will appear which can be written as 05F0

This is the Hexadecimal equivalent of 1520 and if you put &H infront of it the Dragon will treat it in just the same way as 1520. Try typing:-

PRINT &H05F0 + 10 and press ENTER

1530 will appear which is the same as 1520 +10.

I hope you agree because we are in serious trouble if you don't!

Now Hexadecimal numbers are based on 16, not 10 like ordinary numbers, and that means it's very easy to fit them into 16 BITS (2 BYTES). We can now split our Hex number 05F0 in half. The first half, 05, we will call the Most Significant Byte (MSB) because it's in front and the second half, F0, we will call the Least Significant Byte (LSB) because it's at the end. In a M/C programme we can put the operation code (STA) in the first memory location, the MSB (05) in the second memory location and the LSB (F0) in the third memory location like this:-

Memory location 18002 contains STA Extended Direct Addressing
Memory location 18003 contains MSB
Memory location 18004 contains LSB

Now when we POKED these numbers into the Dragon in the last example we converted the MSB and LSB back to decimal numbers. 05 Hex is, by coincidence, 5 in decimal but F0 Hex is 240 in decimal. If you look at the last example you will see that these are the numbers that we used. We could have POKED the Hex numbers in by putting &H in front of them (to tell the Dragon that they were Hex) but that would have looked confusing. If you have not understood this section go back and read it again as it is most important that you understand.

Try converting a few numbers into hex, using the Dragon, and then splitting them into MSB and LSB. Decimal numbers less than 4096 will have less than 4 figures in Hex. Just put zeros in front to make up the difference.

Just think; if computers had 10 BITS in a BYTE how much easier life would be.

As you can see, from the previous example, STA Extended Addressing requires 3 BYTES of memory for the whole operation. The CPU knows this and when it sees Extended Direct Addressing it will automatically look at the next 2 memory locations.

This would be a suitable time to mention the CPU PROGRAMME COUNTER. This is a 16 BIT register in the CPU which keeps track of the instructions to follow. For instance, in the last example, when we typed EXEC 18000 the Programme Register will automatically be loaded with the number 18000. The CPU will go to this address and look at the instruction there. This will tell it to load A with 72 but it will also tell the Programme Counter that the next instruction will be found at memory address 18002. After completing the LDA instruction the CPU will go to memory address 18002 which, as we said before, requires 3 memory locations and the Program Counter will automatically point to 18005. It does this automatically and there is no need for you to set it (unless you want to). It is,however, useful to know what it is doing.

Now back to that AUTO INCREMENT Addressing Mode I mentioned. This uses the X register in the CPU (see page 18).

If we store a 16 BIT memory address in the X register, by using the operation code LDX (similar to LDA), we can tell the CPU to store the contents of the A accumulator at the memory address contained in the X register. For instance:-

```
Load X with 1520
Store A at X
```

This would have exactly the same affect as the last example.

This is known as INDEXED ADDRESSING MODE and can be very useful if we want to keep referring back to a memory address, say to update a score in a game.

A variation on this is AUTO INCREMENT. In this mode we load the X register with a memory address. We can tell the CPU to store the contents of the A accumulator at the memory address in X but this time it will automatically increase X by 1 after completing the instruction. Using the last example the X register would now contain 1521.

In the last program, were we had to store A at different memory addresses, we could have used this addressing mode to do the job for us instead of entering each memory address separately.

This addressing mode is particularly useful if we want to fill the screen with graphic symbols or scan through a set of data, for instance. The next chapter shows an example of this.

To tell the CPU that we want to use Auto Increment we use a POST BYTE. That is a BYTE of information stored in the memory location after Operation Code. It's a bit like having a 2 BYTE instruction instead instead of the usual 1. Calculating Post Bytes is a little tricky so I've dealt with it in detail in the Appendix. Let's use this, plus a few more new instructions, in a program to fill the screen with graphics. Read on.

## A SCREEN DISPLAY

Right, let's write some M/C programmes that will show some screen displays but before we do that we need to look at 2 more instructions:-

CMP and BNE

CMP is used to compare one of the CPU registers or accumulators with something. For instance, CMPA compares the A accumulator with a number. We will be using CMPX which compares the contents of the X register with a number.

BNE means Branch if Not Equal and is usually used with a CMP instruction. For instance:- Compare the contents of the X register with a number and branch if it is not equal to it. It's rather like the Basic IF THEN statement. IF S <> 30 THEN GOTO 100.

Let's try an example. This M/C programme will fill the screen with small red squares:-

1 Load the X register with 1024 (top left of screen)
2 Load accumulator A with 182 (graphic symbol for 2 red squares)
3 Store the contents of A at memory address in X register. Increase X by 1 after doing this.
4 Compare X with 1536 (bottom right of screen)
5 Branch, if it's not equal, to line 3
6 Return to Basic

In this programme we first load the memory address, corresponding to the top left hand corner of the screen, into the X register. We then load the A accumulator with the number of the graphics symbol we want to appear on the screen. Lines 3 4 5 then act as a loop storing he contents of A at each screen memory location ntil it reaches the bottom right corner where he program is complete and control is returned o Basic.

Let's try it. Run the program we used or entering M/C before and enter Start Address 8000. Then enter the following numbers:- (see f you can recognise any of the instructions. 've broken them up into Op code + Data)

```
142 4 0 - 134 182 - 167 128 - 140 6 0 - 38 249
- 57
```

Press break to finish.

Now type EXEC 28000 press enter and ee if you can time how long it takes to fill he screen.

The green stripe and OK prompt appear ecause we have returned to Basic. Our rogramme filled the screen, as we wanted, but hen the green line appeared over the top of it. ever fear we can get rid of it.

The Basic Interpreter is just a very omplex set of Machine Code subroutines to onvert Basic, that we type in, into Machine ode that the CPU can understand. We can use hese subroutines in our M/C programmes. It aves us writing our own. The M/C subroutine we re interested in at the moment starts at memory ddress 32774 (memory map area E page 7) and hat it does is checks the Keyboard to see if a ey is being pressed. Note I said IS being ressed not HAS been pressed. If a key is being ressed it will load the A accumulator with it nd if no key is being pressed it will load the accumulator with zero. It's rather like the NKEY$ function. This is a very useful ubroutine and we will be useing it a lot. I ill illustrate it's operation by adding it to the last programme and then we shall see how it works.

Go back to the M/C entering program we used before and and enter Start Address 28000 then the following numbers:-

```
142 4 0 134 182 167 128 140 6 0 38 249 189 128 6
39 251 142 4 0 126 109 101
```

Press BREAK to finish then EXEC 28000

It's the same as before but with no green line but now press any key and see what happens. Fairly fast isn't it? You try writing a programme in Basic to do that. You will have to press RESET to return control to Basic. You're M/C programme will still be there in memory until you switch off.

Let's see how we did this. We'll look at the first programme. Go back to the explanation. Lines 1 and 2 are straight forward and we have used them before. In line 3 we use the STA instruction in Auto Increment addressing mode. The operation code for this is 167. The Post Byte we use is 128.

The CMPX instruction in line 4 has operation code 140 which is Immediate addressing, meaning that the number it is to compare with (1536) is located in the next memory address. As is it a 16 BIT (2 BYTE ) number it needs 2 memory locations to store it, so the CMPX instruction needs 3 memory locations altogether.

The BNE instruction, in line 5, is rather more difficult. This will cause the program to branch to another memory address if the result of the CMPX instruction is not equal. If the result is equal then the programme will proceed in sequence. The problem comes when we try to work out where the programme is to branch to.

The operation code for BNE is 38 and this will always be followed by another memory address containing the OFFSET. This Offset tells the CPU where to branch to and the way it does it is tied up with the Addressing Mode. The addressing mode used by this instruction is PROGRAMME COUNTER RELATIVE. As we have said before the Programme Counter will contain the memory address of the next instruction the CPU is to carry out. In the case of the BNE instruction, however, the Programme Counter assumes that there will not be a Branch and therefore contains the address of the next instruction in sequence. If the programme does need to branch then the number stored in the Offset will modify the Programme Counter so that it contains the memory address of the instruction you want it to branch to. If, for instance, you want it to branch forward 20 memory addresses then the Offset will contain 20. Conversley, if you want it to branch backwards 20 places then the Offset will contain -20.

Unfortunately it's not as easy as that. For a start you can only branch back 126 places or forwards 129 using this instruction. There is another instruction LBNE (Long Branch if Not Equal) which will allow you to branch farther. Also you must remember, when calculating how far you need to branch, that the Program Counter will not contain the memory address of the BNE instruction but the next instruction, which will be 2 steps on. For instance:- If the BNE instruction is stored in memory address 15023 then the Programme Counter will contain the number 15025.

Calculating the Offset for forward branches is easy. You work out how many steps you want to move forward (from the Programme Counter) and store that number in the memory address following the BNE instruction. To branch backwards you work out how many steps back you want to move then subtract this number from 256. The answer is then stored in the memory address following the BNE instruction as before.

Awkward isn't it? I keep telling you that Machine Code causes severe mental pain but you won't listen otherwise you would have taken up a normal hobby like basket weaving.

To get back to the second programme we just did, where we made the screen fill with whatever key we pressed, you will have noticed that what appeared on the screen was not always the same as the key we pressed. There us a simple answer to this and I will be dealing with it when we look at how the Dragon inputs and outputs information.

The second programme is the same as the first but, after filling the screen with red squares, instead of returning control to Basic we added the following lines:-

<u>6</u> Jump to the subroutine in the Basic Rom starting at memory address 32774. on completion of this subroutine the programme will return and the A accumulator will contain either 0 if no key was being pressed or the value of the key if one was being pressed.
<u>7</u> Check accumulator A, if it contains 0 then branch back to line 6. Go on to line 8 otherwise.
<u>8</u> Load the X register with 1024 (top left of screen)
<u>9</u> Jump back to line 3 (first programme)

We have used 3 new operation codes in this addition to the first programme:-

JSR Jump to Subroutine which is exactly the same as the Basic command. The CPU will jump to a subroutine and continue until it finds a RTS (return from subroutine) instruction. We used Immediate Addressing which means that the memory address of the subroutine to be jumped to follows immediately after the JSR operation code. (2 memory locations required as it's a 16 BIT number.) The whole operation requires 3 memory locations.

JMP Jump which is the same as the Basic GOTO command. The CPU will go to a memory address and carry on in sequence from there. Try and guess what addressing mode we used.

BEQ Branch if Equal to Zero. This instruction caused the programme to branch if the A accumulator contained 0. We can use it a different way which I will explain later but for now we know several instructions so let's see how we use them in programmes.

## PROGRAMMES

We're going to write several short programmes. The idea behind them is that each one will perform a certain function, like clearing the screen, so that we can bolt these together to form more complex programmes. Hopefully that way, even if you don't understand them, you'll be able to use them.

The way I have set them out is:-
On the left hand page will be the programme listing.
On the right hand page will be an explanation.
The programme listing will be set out in 4 columns. The first column will contain the Decimal numbers that you need to POKE into memory using the short Basic programme we have been using.
The second column will contain a line number for reference.
The third column will contain the Mnemonic for the operation code we are using.
The fourth column will contain the Data (if it has a * in front of it) or the memory address (if it has m in front of it) that the Op code will use or an explanation.

At the top of the page will be the programme title and a suggested starting address (this will have to be altered to follow on if you want to string programmes together).

At the bottom of the page will be the last memory address used by the programme.

I have finished most programmes with a RTS instruction to return control to Basic if you want to test each one. In some cases, when bolting programmes together, you may need to delete this. Try experimenting!!

## PROG 1  CLEAR SCREEN

```
Start address 20000

134,128          1    LDA   *128
142,4,0          2    LDX   *1024
167,128          3    STA   X Autoinc
140,6,0,         4    CMPX  *1536
38,249           5    BNE   to 3
57               6    RTS   return

End address  20012
```

A nice easy one to start with but very useful all the same.

We start by loading the A accumulator with number 128 which is the graphics character for a black square. We can put any character here we like. Simply alter the second Byte in the programme.

In line 2 we load the X register with the number representing the memory address of the top left corner of the screen.

In line 3 we store the contents of A at the memory address contained in the X register and also Autoincrement X by 1 thus making it point to the next screen location.

Line 4 compares X with the number 1536 to see if it has reached the bottom right hand corner of the screen yet. It's 1536 not 1535 because after we have stored A at memory address 1535 (bottom right of screen) X will be increased by 1 so the number we want to compare with is 1535+1=1536.

If X does not equal 1536 then line 5 causes it to loop back to line 3.

If X does equal 1536 then line 5 sees that the loop is complete and allows the programme to proceed to line 6 where control is returned to Basic. That's why you have a green line across the screen. If we had stayed in the M/C programme we would have had a nice clear screen.

This was a very simple programme to fill the screen with any graphics character you like, or it can be a clear screen instruction. Now let's put some stripes on it:-

## PROG 2 STRIPES

Start address 20012

```
134,239          6    LDA   *239
142,5,0          7    LDX   *1280
167,129          8    STA   X Autoinc x2
140,5,32         9    CMPX  *1312
38,249          10    BNE   to 8
57              11    RTS   return
```

End address 20024

By making the start address 20012 we will overwrite the RTS instruction, of the last programme, and replace it with the first instruction of the second programme.

Line 6 now loads A with the graphics character 239 and line 7 sets the first memory address, we will store this character at, as 1280 which is about halfway down the screen on the left.

Line 8 stores the character, as in Prog 1, but this time we Autoincrement by 2. This means that X now becomes 1282 and the graphics character will be displayed every other space on the screen.

As well as Autoincrementing by 1 or 2 we can Autodecrement, or decrease X, by 1 or 2 (see Appendix A). There is a slight difference between Autoinc and Autodec though. If we STA using Autoinc we store A and then increase X by 1 or 2 but in Autodec we reduce X by 1 or 2 and then store A. Be careful with this as it is easy to get confused.

The rest of the programme is the same as the last finishing with a RTS instruction which will return us to Basic and also leave us with a green stripe.

Now we have a horizontal stripe, that we can put anywhere we like by altering the start and finish memory locations, let's try a vertical stripe.

## PROG 3 VERTICAL STRIPES

Start address 20024

```
134,159          11    LDA  *159
142,4,31         12    LDX  *1280
167,132          13    STA  X
48,136,32        14    LEAX X+32
140,6,31         15    CMPX *1567
38,246           16    BNE  13
57               17    RTS  return
```

End address 20039

Again we have overwriten the RTS of the last programme and also loaded the A accumulator with a graphics character and loaded X with the memory location where we want our display to start (top right corner).

This time the STA instruction uses REGISTER ADDRESSING. This means that A will be stored at the memory address contained in the X register. We use a Post Byte to indicate this (see Appendix B) and in this case the Post BYTE is 132.

We now come to the LEAX instruction which means Load Effective Address. This will load the X register(last letter of op code. LEAY loads Y register for instance) with the contents of another register. In this case we are using an addressing mode that allows us to add a number (32) to the X register. We do this by loading X (LEAX) with the contents of X plus 32 (or any other number).

In lines 15 & 16 we do our normal compare and branch routine which allows the LEAX instruction in line 14 to increase X by 32 every time. As there are 32 Characters to a line this means that we will store A at the end of each line. If we only increased X by 30 (LEAX X+30) we would have a diagonal line.

The LEA instruction is a very powerful tool and we shall be using it again. Try experimenting by altering the last Byte in line 14.

I will now show you how to scan blocks of memory and display them on the screen. This is useful for writing titles or searching for information.

## PROG 4   WRITE TITLE

```
Start address 20039

142,4,168        17    LDX   *1192
16,142,78,91     18    LDY   *20059
166,160          19    LDA   Y Autoinc
167,128          20    STA   X Autoinc
16,140,78,103    21    CMPY  *20071
38,246           22    BNE   19
57,18,18         23    RTS   and NOP
80,82,69,83,     24          DATA
83,32,89,32,79,82,32,78

End address 20070
```

Things now become a little more complex. Line 17 loads the X register with the memory location of the place on the screen where I want to start displaying my message.

Line 18 loads the Y register with the address in memory where I am storing the information I want to display.

We now start to scan through that information in line 19 by loading the A accumulator with the information contained in the memory address pointed to by Y. We will also Autoinc Y so that we can scan through this information.

Line 20 stores this information on the screen and Autoincs X so that the information is displayed across the screen.

Line 21 compares Y to see if we have reached the end address of the stored information. If we want to put a longer message in we simply increase this number.

Line 22 causes the programme to loop until all the information is read and displayed.

Line 23 returns to basic and also contains 2 NOP (No Operation) codes that are there to fill in space that we will use in the next programme.

Line 24 is the actual data that we are storing and is the ASCII codes for PRESS Y OR N. We could store this information anywhere in memory if we wished but we must Load Y with its starting address in memory and then compare Y with its end address so that the routine can scan these memory addresses.

We can use a routine like this to scan any area of memory to look for information or.

## PROG 5   Keyboard Scan

Slight change here:-

```
POKE 20056,126
POKE 20057,78
POKE 20058,103
```

This tells the CPU to Jump to memory address 20071. This is so it does not look at the data we stored, starting at 20059, and think they are instructions.

Start address 20071

```
189,128,6        25    JMP   m32774
39,251           26    BEQ   25
129,89           27    CMPA  *89 (ASCII Y)
38,3             28    BNE   30
126,78,122       29    JMP   33
129,78           30    CMPA  *78 (ASCII N)
38,112           31    BNE   25
126,78,128       32    JMP   35

183,5,72         33    STA   m1352
126,78,131       34    JMP   36
183,5,83         35    STA   m1363

57               36    RTS   return
```

We start by POKEING a jump into the last programme so that the CPU does not look at the data we stored and think it's instructions.

Line 25 then jumps to a subroutine in the Basic Rom starting at memory address 32774. This scans the keyboard to see if a key is being presses. If a key is being pressed its value will be loaded into the A accumulator and if not zero will be loaded.

Line 26 checks to see if A contains zero and if it does it loops the programme back to line 25 to start again.

Lines 27 to 32 check to see if either Y or N have been pressed. If not then the programme loops back to 25 to start again. If Y is pressed the programme jumps to line 33 which causes the letter Y to be displayed on the screen. If N is pressed then the programme will jump to line 35 and N will be displayed.

This is a very useful programme because not only does it check the keyboard for input but it also allows you to make decisions. You can jump to anywhere in memory simply by altering the memory addresses (last 2 Bytes) in lines 29 or 32.

Let's now look at scrolling the screen. There have been a lot of programmes in magazines that scroll the whole screen up and down or side to side so I thought I'd do something different.

## PROG 6 SCROLLING

Start address 20099

```
142,4,160       36    LDX   *1184
166,128         37    LDA   X Autoinc
183,4,191       38    STA   m1215
166,132         39    LDA   X
48,31           40    LEAX  X-1
167,132         41    STA   X
48,2            42    LEAX  X+2
140,4,192       43    CMPX  *1216
38,243          44    BNE   39
48,1            45    LEAX  X+1
140,39,16       46    CMPX  *10000
38,249          47    BNE   45
126,78,131      48    JMP   36
```

End address 20129

Well that's the last programme for this section. Try and work out how it works yourself. Line 40 subtracts 1 from the X register and line 42 adds 2 to X. We could have used Autoinc and Autodec here but I wanted to show the use of the LEAX instruction to add or subtract from registers.

Lines 45,46,47 are a simple delay loop otherwise the screen would move to fast. X is increased by 1 each pass of the loop until it reaches 10000 and then the programme returns to line 36. This gives you some idea of how fast the CPU works. If you used basic in a FOR NEXT loop to count to 10000 it would take very much longer.

You can now use these programmes as a basis for building up quite long routines. Try and experiment with them, it's the best way to learn. The only thing you can lose is your temper!!

All the programmes, so far, have used low resolution graphics. We will now look at how to use high resolution and the sound facilities of the Dragon.

## SAMUN PIE

If you look at the memory map on page 7 you will see an area from memory address 65280, called G, labeled System Use. This area contains the SAM and PIA chips (Samun Pie, sorry about that). These are the silicon chips that control the computer and handle its Input and Output. Although they are silicon chips it is easier to imagine them as a set of memory locations. We will look at them in turn.

First we will look at the PERIPHERAL INTERFACE ADAPTERS (PIAs). These handle communication between the CPU and the outside world ie. the Keyboard, TV, Cassette Recorder etc. If we imagine these devices as memory addresses the ones we are interested in are 65280 to 65283 and 65312 to 65315. 8 memory addresses altogether and it does not seem many to handle all that input and output. The way it does it is rather clever. Look at the diagram on the next page. There are 2 PIA chips but each one contains 2 input/output ports so I like to think of them as 4 separate devices. They are normally Known as:-

PIA 0 A, PIA 0 B, PIA 1 A, PIA 1 B

As you can see each of these devices has 2 memory addresses and 3 internal 8 BIT (1 BYTE) registers. These internal registers are known as:-

| | |
|---|---|
| CONTROL REGISTER | (CR) |
| DATA DIRECTION REGISTER | (DDR) |
| PERIPHERAL DATA REGISTER | (PDR) |

Only the Americans could have thought of names like that! I've never quite seen the point of describing things with very long words and then having to shorten them but I digress.

## PERIPHERAL INTERFACE ADAPTER

| | PIA 0 | PIA 1 | |
|---|---|---|---|
| A | 65280 | 65312 | Peripheral data register<br>Data direction register |
| | 65281 | 65313 | Control register |
| B | 65282 | 65314 | Peripheral data register<br>Data direction register |
| | 65283 | 65315 | Control register |

FIG 3

PIA CONTENTS

| it No. | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| 5280 | Keybord Row | | | | | | Joystick | |
| 5281 | | | | | | | | |
| 5282 | Keyboard Column | | | | | | | |
| 5283 | | | | | Snd | | | Int |
| 5312 | Digital to Analogue | | | | | | Pnt | Cas |
| 5313 | | | | | Mot | | | |
| 5314 | Video Lines | | | | | | Snd | |
| 5315 | | | | | Snd | | Cartridge | |

The DDR and PDR share a common memory address. How the CPU selects which one to use I will explain in a moment.

Each register consists of 8 BITS and we will consider then as 8 separate switches that can be set on or off independently. The 8 BITS are numbered as shown:-

| Data-BIT | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |

By putting a BINARY number in these BITS we can set each BIT either ON with a 1 or OFF with a 0. For instance, the number 251 converted to a Binary number is 11111011. If we store this number in memory address 65281 it will switch on all the BITS except number 2 which will be switched off. By doing this we can make each BIT responsible for a particular function and by simply storing different numbers in this memory location we can switch on or off up to 8 separate functions.

The Control Register (CR) will decide which of the other 2 registers can use the memory address they share. If Bit 2 is set to 0 the DDR is selected and if BIT 2 is set to 1 the PDR is selected.

The Data Direction register selects whether each BIT is set as an Input or Output. Storeing a 0 sets it to Input and 1 sets it to Output. Once these Bits have been set, by the computer at switch on, control is usually handed back to the Peripheral Data register.

Fig 3 shows each of the registers along with their normal contents immediately after the computer has been switched on.

The PDR at memory address 65280 is concerned with the Keyboard Row input and also the Joysticks. A 0 in BIT 1 means that the left Joystick is being fired and a 0 in BIT 0 means that the right Joystick is being fired. These Bits are also shared with the Keyboard Row inputs (which row of keys is being pressed) and explains why in, some games, pressing keys has the same effect as the fire button. When a key is pressed a 0 appears in the BIT corresponding to the keyboard row that was pressed.

The only BITS we are interested in in the CR at memory address 65281 are BIT 2 which controls whether the DDR or PDR is selected and BIT 3 which is set to 0 when the sound is enabled.

PDR at 65282 is the Keyboard Column input. When a key is pressed BITS are set in this register corresponding to the column of the key being pressed. As this memory address shows the column and 65280 shows the row the computer can tell exactly which key, or combination of keys, is being pressed. However, the computer clears this register as soon as it has made a note of its contents so if you PEEK (65282) you will always get a 0. The information is stored, however, in the bottom end of memory and we will be looking at this in the next chapter.

CR at 65283 uses BIT 2 to select DDR or PDR as before. BIT 3 is set to 0 when the sound is enabled and Bit 0 controls the Interrupt sequence. The CPU refers to this location 50 times a second as part of the TIMER and PLAY function. If you set this bit to 0 Play commands will go on for ever and the TIMER will halt. This is a good way to cheat at games that have a time limit.

The PDR at 65312 is most useful. BIT 0 is the Cassette Input and BIT 1 is used by the

Printer. BITS 2 to 7 are used as a Digital to Analogue converter which means that when the Sound is enabled feeding numbers into these bits will produce sound so we can use them to play music.

CR at 65313 BIT 2 selects DDR or PDR as usual. BIT 3 controls the cassette remote. A 1 switches it on and a 0 switches it off. We used this at the very beginning of this book.

The PDR at 65314 BITS 3 to 7 in conjunction with the SAM chip control the video display. We will be discussing these in detail later and explaining how they decide what graphics mode we are in (PMODE 4,1 etc). BIT 1 is used in conjunction with the digital to analogue to produce sound.

CR at 65315 again uses BIT 2 to select DDR or PDR. BIT 3 is the sound enable. When this BIT is set to 1 sound is outputed to the TV (BITS 4&5 must also be 1). BITS 0 and 1 are used to Autostart Cartridges if they are present. You can stop a cartridge autostarting by setting BIT 0 to 0 before inserting the cartridge but I would strongly advise against plugging in cartridges when the Dragon is switched on as I already have one friend who ruined his machine doing this.

What all this means is that we can control various input/output features of the Dragon by storing the correct number at these addresses. In practice the only ones we will use are the sound, graphics and joystick ones.

The SYNCHRONOUS ADDRESS MULTIPLEXER or SAM chip is another clever device. If the CPU is the brains of the Dragon then the SAM directs the traffic. There is a lot of data flying about in the Dragon and lots of devices all wanting to work at the same time. The SAM chip keeps everything in the correct order.

We will think of this device, once again, as a set of memory addresses. The SAM has a set of functions that we are interested in. Each function has 2 memory addresses ie. 65494 and 65495. If we store any number (it doesn't matter what) in the odd numbered address (65495) we will switch the function on and if we store any number in the even numbered address (65494) we will switch it off. Fig 4 shows the functions that we will be interested in. Don't forget, even number addresses switch off - odd number addresses switch on.

VDG MODE addresses 65472 to 65477 in conjunction with the PIA at memory address 65314 control which Graphics Display Mode the computer uses ie. PMODE 4 or normal screen for instance. There are 3 functions in the SAM to set and 5 BITS in the PIA for each display mode. Fig 4 shows how each device should be set for each mode. On switch on or reset the computer automatically selects the Standard screen display.

When these devices are set they control a device called a VIDEO DISPLAY GENERATOR (VDG). This device will go to the area of memory reserved for graphics display, look what is stored there and turn it into a display to be sent out to your TV. The VDG has its own set of internal characters that it can display. To see these POKE 1200,(numbers from 0 to 255) and they will be displayed on the screen. You will notice these do not quite tie up with the normal ASCII codes.

Addresses 64578 to 65491 control where in memory the VDG starts to look for the screen display. This will normally be set to memory address 1024 but can be different if, for instance, you are looking at different PAGES of

FIG 4

## SYNCHRONOUS ADDRESS MULTIPLEXER

| Address | | | |
|---|---|---|---|
| 65501 | | | Memory Size 4k 16k 64k |
| 65498 | | | |
| 65497 | s | R1 | CPU Speed |
| 65496 | c | | |
| 65495 | s | R0 | |
| 65494 | c | | |
| 65491 | | | Start Address for Display |
| 65478 | | | |
| 65477 | s | V2 | Video Mode |
| 65476 | c | | |
| 65475 | s | V1 | |
| 65474 | c | | |
| 65473 | s | V0 | |
| 65472 | c | | |

s= Set or On

c= Clear or Off

graphics.

Addresses 65494 to 65497 control the speed that the CPU works at. Normally the CPU works at a speed of 8000 cycles per second if R0 and R1 are off. If however you switch on R0 by storing a number at address 65495 you will double its speed. 2 things happen here. Firstly the Dragon was designed to run at a slow speed but if you are lucky you will have one that is at a higher than standard specification and it will run quite happily at this higher speed. If your system refuses to function when you use this command you're not one of the lucky ones. Secondly the CPU only works at double speed over half of the memory map. The part used by the basic interpreter and the cassette input output. This means it will speed up your basic but you won't be able to load or save any programmes until you switch R0 off.

If you switch R1 on the CPU will work at double speed over the entire memory map and this has to be seen to be believed. Unfortunately the VDG cannot function at this speed so your TV display will go haywire but if you have a large number a figures to handle, in a Basic programme, switching on R1 at the start and switching it off at the end will really speed up your program. The TV display will return to normal as soon as you switch R1 off.

That's a brief outline off the top of your computers memory map. Now let's look at the bottom.

## THE BOTTOM END

At the bottom end of the Dragons memory map is an area from 0 to 1023 reserved, by the computer, for its own use. In it the CPU stores information that it will need whilst running programmes. For instance, at memory address 135 the CPU stores the value of the last key that was pressed and in memory address 182 is stored the current graphics mode. Although the CPU controls this section we can either inspect its contents or change them to suit our own programmes. The best way I found to inspect this area is to POKE and PEEK into it and see what happens. The results can be quite spectacular so make sure you don't have any important programmes stored in memory because they probably won't be there when you've finished.

For those of you who are more faint hearted I have listed some of the locations that I know off:-

Most of these locations contain memory addresses so they are 2 BYTES long. The MSB is stored in the lowest location number and the LSB in the highest.

For instance, to find the highest memory available to Basic programmes type the following:-

```
PRINT PEEK (39) * 256 + PEEK (40)
```

39 & 40 Highest memory space available to Basic programmes.
25 & 26 Beginning of Basic programmes.
31 & 32 End of Basic programme.
68 & 69 Current line number in Basic programme.
111 Current device being used ie. 0 = VDU, -1 = cassette, -2 = printer

| | |
|---|---|
| 20 | Is cassette being used ie. 0 = no, 1 = nput, 2 = output |
| 35 | Value of last key pressed. |
| 78 | Foreground colour. |
| 179 | Background colour. |
| 182 | Current graphics mode. |
| 183&184 | Start address of current graphics screen. |
| 86&187 | End address of current graphics screen. |
| 85 | Number of Bytes in a row of graphics. |
| 89&190 | X position. |
| 91&192 | Y position. |
| 276 | Timer, cycles from 0 to 255 in about 5 seconds. |
| 329 | Sets Inverse graphics (same as SHIFT 0) ormally 255 POKE with 0 for inverse. |
| 36&137 | Address of next location on screen. |
| 55 | Length of a line when outputing to rinter (normally 80 characters). |
| 56 | Position of printer head on line. |
| 487&488 | Start of machine code programme after loading. |
| 126&127 | End of machine code programme + 1 subtract 1 for actual address). |
| 157&158 | EXEC address of M/C programme. |

These are just a few of the addresses hat are useful. I hope you can find some more.

## MORE PROGRAMMES

We now have an idea of how the Dragon uses its Input/Output and SAM chip. Let's see how we use them in practice. I shall use the same programme layout as before but this time I will assume that we are going to use some of the programmes from within a Basic programme. There will, therefore, be a lot of Peeks and Pokes.

You may be wondering why I've not used the High Resolution screen modes so far. Well there's a very simple reason for this. It's a real pain. It's a lot easier, if you want fast moving graphics in your Basic programme, to get into the PMODE that you want and draw your shapes using Basic and then use a short machine code programme (like Prog 6) to move the screen quickly.

But for those of you who insist on struggling here is how you get into PMODE 4:-

First we need to set the SAM chip VDG lines. We switch V0 off, V1 on, V2 on. This is done by storing any number at the following memory addresses:- 65472 65475 65477 Infact we cannot actually store anything at these addresses as the SAM chip has no where to put anything. It can, however, detect that something is trying to address it and it takes the appropriate action. That's why we can store any number we like.
You can either use the STA m65472 instruction in M/C or Poke 65472,0 in Basic

After setting the Sam chip we then need to set the PIA chip. We do this by storing the number 253 ( or 245) at memory address 65314. Again we can use either the STA operation code or POKE 65314,253 (or 245).

We are now in PMODE 4 but the high

resolution mode uses memory locations 1536 to 7167 to store its information so we must again set the SAM chip so that it looks at these locations and not 1024 to 1535. (These locations for the normal screen are set automatically every time the computer is switched on.) So we must store any number in the following memory addresses:- 65479 65481 65482 65484 65486 65488 65490

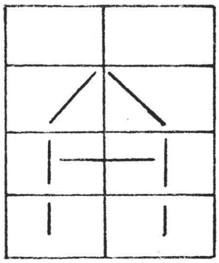
We are now in PMODE 4 and will stay there until something allows the programme to return to Normal screen. ie you return to Basic. Tricky isn't it?

A much easier trick is to use the Semigraphic mode 24. You won't find anything about this in the Dragon manual. To get into this mode we must store a number in memory addresses 65472 65475 65477

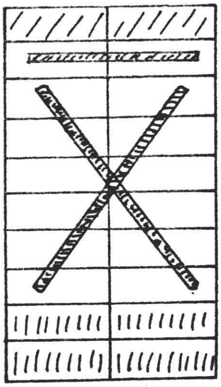
Mode 24 uses memory locations 1024 to 7167, which is the same start address(1024) as the normal screen so we don't need to set the SAM again. We can also ignore the PIA for the moment. As you can see this is a much easier mode to get into.

In this mode the screen is divided into 32 spaces across by 192 spaces down. The top left hand corner uses memory address 1024 and the bottom right corner uses memory address 7167. To see how we put information onto this screen we first need to look at how a graphics character is made up.

Look at the diagram on the next page. Characters are made up from a block of Pixels 2 x 4. In the normal screen mode this is displayed as a whole block but in mode 24 it displays it one row at a time. If we want to put a red line on the mode 24 screen we just store in each memory address the code number for the graphics block which has a top line that is red. ie 191. We can make up quite complex shapes using this method.


Letter A

The simplest way to understand this is to try it so let's make a row of space invaders appear. They will be made up of blocks as shown in the diagram and we will just repeat this at every other location across the screen.


Space Invader

We will write a short programme to display a row of characters, that are supposed to look like space invaders, across the Mode 24 screen. First we will get into mode 24 and then clear the screen. We will then place our invaders in a row across the middle.

# PROG 7 SPACE INVADERS

```
Start address 22000

183,255,192        1    STA   m65472
183,255,192        2    STA   m65475
183,255,195        3    STA   m65477

134,128            4    LDA   *128
142,4,0            5    LDX   *1024
167,128            6    STA   X Autoinc
140,28,0           7    CMPX  *7168
38,249             8    BNE   6

142,16,32          9    LDX   *4128
134,191            10   LDA   *191
167,129            11   STA   X Autoinc x 2
140,16,64          12   CMPX  *4160
38,249             13   BNE   11

134,20             14   LDA   *20
167,129            15   STA   X Autoinc x 2
140,16,128         16   CMPX  *4224
38,249             17   BNE   15

134,24             18   LDA   *24
167,129            19   STA   X Autoinc x 2
140,17,64          20   CMPX  *4416
38,249             21   BNE   19

134,159            22   LDA   *159
167,129            23   STA   X Autoinc x 2
140,17,128         24   CMPX  *4480
38,249             25   BNE   23

189,128,6          26   JSR   m32774
39,251             27   BEQ   26
57                 28   RTS   return

End address 22065
```

Lines 1 to 3 set the SAM chip to mode 24. Lines 4 to 8 are used to clear the screen, as before, by storing graphics character 128 (black block) at every screen location.

Lines 9 to 13 store the graphics character 191 (red block) at every other screen location between 4128 and 4159, which will go in a line across the approximate middle of the screen.

In lines 14 to 17 we use the graphics character T (reversed) to put a line across the screen. This will only print the top 2 pixels so in fact we get a - not the full T. See the diagram on the preceding page.

We now come to the clever bit. Lines 18 to 21 store the graphics character X (reversed) on the next 6 of the 192 lines on the screen. What this does is to display the bottom part of the X. It will appear to have the top of the T mixed with it and is easier to see than describe.

Lines 22 to 25 put two yellow lines underneath.

You can see that we can make all sorts of shapes by mixing lines of characters but there is a snag. The characters will appear on the screen in the position that they would be in if in the normal mode. What this means is that we must try and work out what part of the character would be displayed at the position we are going to put it at on the screen. What I have done is to take a copy of the PRINT @ GRID in the Dragon manual and draw 192 lines across it. This makes it a little easier to calculate.

If you still find this difficult to grasp add 64 to all of the numbers in lines 9,12,16,20,24. This will have the effect of moving the display down 2 lines and should illustrate what I mean.

We did not need to keep loading the X register with a new value every time we printed a new line because, if you work it out, it already has the correct memory address, for the start of the line, in it.

This display can be moved from side to side using a routine similar to PROG 6. There have been dozens of programmes published in magazines showing how to move graphics screens up and down or side to side so I won't go into them. I always recommend people to study other peoples programmes. It's the best way to learn.

We can now look at making sounds come out of the TV which is fairly easy.

Switching on the Audio is the easiest way to start. This is the same as the Basic command AUDIOON. It is controlled by the PIA at memory location 65315. If you Peek this you will find the number 55 stored in it. We need to set bit 3 of this location (refer back to the chapter showing the PIA layout).

We can do this by storing the number 63 at this address either by using the STA instruction, in a M/C programme, or by POKE 65315,63.

We also need to set Bit 3 of 65281 by storing 188.

That was fairly simple but to make sounds we need to go a little further.

The sound output is affected by the contents of the PIA at memory address 65312. This is the Digital to Analogue converter. But first we need to set 65315 65281 & 65283. The following program shows how altering the contents of 65312 affects the sound output.

## PROG 8    SOUND

Start address 23000

```
182,255,35          1       LDA   m65315
138,8               2       ORA   *8
183,255,35          3       STA   m65315

182,255,1           4       LDA   m65281
132,247             5       ANDA  *247
183,255,1           6       STA   m65281

182,255,3           7       LDA   m65283
132,247             8       ANDA  *247
183,255,3           9       STA   m65283

134,20              10      LDA   *20
183,255,32          11      STA   m65312
76                  12      INCA
129,255             13      CMPA  *255
38,248              14      BNE   11
32,244              15      BRA   10
```

End address 23035

Lines 1 2 & 3 take the contents of 65315, whatever it is, and make sure that bit 3 is on. It does this by using a Logical OR instruction, which I will explain on the next page. Line 3 then places it back in memory address 65315. What we are in fact doing is to make sure that Bit 3 in memory address 65315 is set on without altering any of the other Bits.

Lines 4 5 & 6 and 7 8 & 9 do the same thing to memory addresses 65281 and 65283 but this time we make sure Bit 3 is set off. In this case we do this by using the Logical AND instruction which I will also explain on the next page.

From line 10 onwards we load A with 20 which we then store in the Digital to Analogue converter in the PIA at memory address 65312.

We then increase the number in A by 1 and loop back to line 11 to store it again. This then load the Digital to analogue converter with an increasing number. When A reaches 255 it loops to line 10 where it is set to 20 and the whole process repeats.

I'm afraid you will have to press the reset button to stop it. In a normal programme you would have this continuing for a certain time.

Try altering the number 255 in line 13 and number 20 in line 10.

The logical OR and AND instruction are very useful for altering certain Bits of a binary number without altering the rest. An illustration is the best thing to show you.

```
            110111   = 55
         OR 001000   =  8

    Becomes 111111   = 63
```

Bit 3 has been Set to on

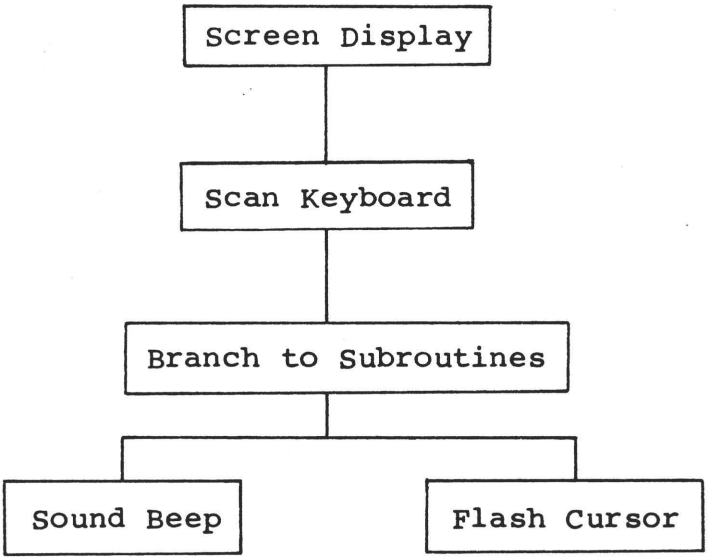
```
            10111100   = 188
        AND 11110111   = 247

    Becomes 10110100   = 180
```

Bit 3 has been set to off.

## WRITING PROGRAMMES

If there was an easy way to write M/C programmes I'd market it and make a fortune. Unfortunately there's not so we have to do it the hard way. Sitting down with a pencil and paper and writing down what you want to do first is the best way. When you Know what you want to do break it down into small logical steps. I use a flow diagram like this:-



When you have reached this far write short M/C routines for each part then string them together. I like to use lots of subroutines but you have to be careful to keep track of where you are.

The easiest thing to do is write your programmes in basic and use short M/C routines for things like fast scrolling of the screen. I don't bother with the USR0 function instead i just write an EXEC (whatever address the machine ode is at) and make sure my M/C ends with a RTS p code.

You can put your M/C in as part of your asic programme using a DATA statement to store he Op Code numbers then just READ them and POKE hem into memory as part of the first few lines f your programme.

## ASSEMBLERS AND DISASSEMBLERS

We can now look at a very useful device called an Assembler, sometimes known as an Editor Assembler. It is similar to the Dragon Basic Interpreter in that it converts the language that we type on the screen into machine code that the CPU can understand so it must be a machine code programme that we can load into the Dragon.

What it does is to take the operation code mneomi, that we have been using (LDA STA etc), and converts them into the operation code numbers that the CPU uses. This means that we do not have to remember all the op code numbers, which is a good idea for a start, but it does far more than this. It can work out what addressing mode we are using which I find a tremendous help. But the best thing, I find, is its ability to work out all your branch and jump instructions without you having to work out Post Bytes and Offsets.

You can get Assemblers either on cassette or cartridge. I personally use the MACE editor assembler, on cartridge, which I find very good. I use a cartridge because it leaves a lot of memory free to develop long M/C programmes and also I can leave it permanently plugged in to be called up when I need it. However, cartridges are expensive and unless you are going to do a lot of serious programming I would get an Assembler on tape for about a third of the price. All Assemblers have different methods of working but let's take a broad look at how to use them.

When you use your assembler what you type in will appear on the screen in 4 separate columns or FIELDS (This is the term commonly used). In the first Field (column) you can put a LABEL that the assembler can use as a reference point. ie Start would indicate the start of the programme but can also be used as a reference point for a Branch instruction.

Example:- BNE to Start

This saves us having to work out the offset as the assembler does it for us. You do not have to put a Label in the first Field (column), it is optional.

You must put something in the second Field as this is used to hold the Operation Code Mneomi (LDA STA etc). The assembler will convert the mneomi into the operation code number that the CPU will recognise.

The third Field is used to store the OPERAND or number/memory address that the operation code is to use. You must always have a number in this Field also.

The fourth and final Field is used to store remarks and comments. The assembler ignores this column and it is used to store comments and memory joggers for the programmer.

When typed in an assembler program will look something like this:- The first line contains a line number for reference.

```
    FIELD1 FIELD2 FIELD3 FIELD4

001 START  LDA    *157   Start of programme
002        STA    1024   Start of screen
003        JMP    START  Return to begginin
```

After we have typed the programme in like this we then tell the Assembler to assemble or convert this into M/C and place it in the Dragons memory. We then have a full working M/C programme without most of the difficult working out of Offsets etc.

I said a full working M/C programme but as we all know computer programmes very rarely work first time. This is were the EDITOR part of the assembler comes in. Useing this we can examine and alter our programme to try and debug it. How easy this is depends on how good your assembler is and what sort of mess you have made of the programme.

As well as the normal Operation Code Mneomic there are certain commands that only the assembler recognises. These are things like ORG or origin which tells the assembler where you want it to store the programme in memory.

The Editor Assembler is a very powerful programme and you will need one if you wish to progress in M/C programming.

The DISASSEMBLER does exactly the opposite. It is another programme that looks at a machine code set of operation code numbers and converts them back to mnemomics. The disassembler is very useful for looking at other M/C programmes that you wish to examine. You can purchase Disassemblers on cassette and there seems to have been a glut of them published in magazines just lately. I must admit though that I haven't seen one that really impresses me and after disassembling a programme you will still have a lot of head scratching to do in order to work out what program flow the original programmer had in mind. Still that's half the fun.

One problem I find with disassemblers is that they are usualy fairly long and take up a fair bit of memory so that when you load in the M/C programme you wish to examine you usually write over the top of your disassembler and the results can be quite remarkable.

What I do is to check where in memory my disassembler is and then load in the M/C programme into a safe area of memory using offset loading (see Dragon manual page 135).

# ODDS AND THE ENDS

I've left this section for all the various odds and ends that I thought about after writing the rest of the book. For instance:- Tandy Colour Computer M/C programmes can be converted to run on the Dragon fairly easily. The Tandy Colour Computer is almost exactly the same as the Dragon. Main areas of difference are in the Basic Assembler Rom. Try looking for JSR operation codes in your Tandy programme. The Tandy uses a routine starting at address 40970 to check the keyboard. Try changing this to JSR 48101. JSR 40970 checks the Joysticks (try JSR 48466). JSR 40962 outputs a character to a device (try JSR 48299). The Rainbow magazine (if you can find one) is a good source of information.

To save a machine code programme on the Dragon use the command:-

```
CSAVEM"program name",Start address, End address,
Program entry address
```

Start and end addresses are where your programme starts and finishes in memory. Programme entry address (not difference between start and finish as stated in the Dragon manual) is the address at which your programme starts and is usually the same as the start address. Some programmes start in the middle and then jump back to the start address.

To find out where a M/C programme is in memory type the following after loading your programme:-

```
PRINT PEEK(487)*256+PEEK(488) ENTER =start

PRINT PEEK(126)*256+PEEK(127)-1  ENTER =end
PRINT PEEK(157)*256+PEEK(158) ENTER =entry
```

If you want to examine a self start programme on cassette then try loading it with an OFFSETT ie:-

```
CLOADM"programme name",1000
```

That will load it into memory 1000 BYTES higher than it should be and should cure the problem. It won't work there but you will be able to examine it by PEEKING.

Well that about wraps it up. You should now know a little bit about machine code. I hope you've found it interesting. There is a lot more to learn. We've only looked at about half of the commands and addressing modes. If you want to go deeper into the subject try reading the following:-

6809 Assembly Language Programming by Lance A Leventhal is the best but a little expensive.

THE 6809 COOK BOOK is also good and cheaper.

6809 MICROCOMPUTER PROGRAMMING & INTERFACING with experiments is good for the more practical experimenters.

And now you can go back and read again all those magazine articles about M/C. You may understand them now. And don't forget, the best way to find out about machine code is to experiment. GOOD LUCK.

## COMMON OPERATION CODES

Decimal Op codes for different address modes

| | INH |
|---|---|
| ABX | 58 |

Adds the contents of A to X and store in X

..................................

| | IMM | DIR | EXT | IND |
|---|---|---|---|---|
| ADDA | 139 | 155 | 187 | 171 |
| ADDB | 203 | 219 | 251 | 235 |
| ADDD | 195 | 211 | 243 | 227 |

Adds contents of memory to contents of accumulator specified by last letter

..................................

| | IMM | DIR | EXT | IND |
|---|---|---|---|---|
| ANDA | 132 | 148 | 180 | 164 |
| ANDB | 196 | 212 | 244 | 228 |
| ANDCC | 28 | | | |

Logically ANDs the contents of memory with specified accumulator

..................................

| | REL |
|---|---|
| BEQ | 39 |

Branches if zero

..................................

| | REL |
|---|---|
| BHI | 34 |

Branches if higher

..................................

| | REL |
|---|---|
| BLO | 37 |

Branches if lower

..................................

|  | REL |
|---|---|
| BNE | 38 |

Branches if not equal

...............................

|  | REL |
|---|---|
| BRA | 32 |

Branch always

...............................

|  | REL |
|---|---|
| BSR | 141 |

Branch to subroutine

...............................

|  | INH | DIR | EXT | IND |
|---|---|---|---|---|
| CLR |  | 15 | 127 | 111 |
| CLRA | 79 |  |  |  |
| CLRB | 95 |  |  |  |

CLR loads memory with zero
CLRA/B loads accumulator with zero

...............................

|  | IMM | DIR | EXT | IND |
|---|---|---|---|---|
| CMPA | 129 | 145 | 177 | 161 |
| CMPB | 193 | 209 | 241 | 225 |
| CMPX | 140 | 156 | 188 | 172 |

Compares with specified accumulator/register

...............................

|  | INH | DIR | EXT | IND |
|---|---|---|---|---|
| DEC |  | 10 | 122 | 106 |
| DECA | 74 |  |  |  |
| DECB | 90 |  |  |  |

Decreases the contents of memory or accumulator by 1

...............................

|  | INH | DIR | EXT | IND |
|---|---|---|---|---|
| INC |  | 12 | 124 | 108 |
| INCA | 76 |  |  |  |
| INCB | 72 |  |  |  |

Increases the contents of memory or accumulator by 1

..............................

|  | DIR | EXT | IND |
|---|---|---|---|
| JMP | 14 | 126 | 110 |

Jump to memory address

..............................

|  | DIR | EXT | IND |
|---|---|---|---|
| JSR | 157 | 189 | 173 |

Jumps to subroutine at specified memory address

..............................

|  | IMM | DIR | EXT | IND |
|---|---|---|---|---|
| LDA | 134 | 150 | 182 | 166 |
| LDB | 198 | 214 | 246 | 230 |
| LDD | 204 | 220 | 252 | 236 |
| LDX | 142 | 158 | 190 | 174 |

Loads accumulator/register

..............................

|  | INH |
|---|---|
| MUL | 61 |

Multiplies A by B place result in D

..............................

|  | INH |
|---|---|
| NOP | 18 |

No operation does nothing but increase programme counter by 1

..............................

|  | IMM | DIR | EXT | IND |
|---|---|---|---|---|
| ORA | 138 | 154 | 186 | 170 |
| ORB | 202 | 218 | 250 | 234 |
| ORCC | 26 |  |  |  |

Logically ORs the contents of memory with specified accumulator

..............................

|  | INH |
|---|---|
| RTS | 57 |

Returns from subroutine

..............................

|  | DIR | EXT | IND |
|---|---|---|---|
| STA | 151 | 183 | 167 |
| STB | 215 | 247 | 231 |
| STD | 221 | 253 | 237 |
| STX | 159 | 191 | 175 |

Stores contents of accumulator/register in memory

..............................

|  | IMM | DIR | EXT | IND |
|---|---|---|---|---|
| SUBA | 128 | 144 | 176 | 160 |
| SUBB | 192 | 208 | 240 | 224 |
| SUBD | 131 | 147 | 179 | 163 |

Subtracts memory from specified accumulator/ register

..............................

|  | INH |
|---|---|
| TFR | 31 |

Transfers contents of A to B

..............................

# APPENDIX B

## POST BYTES

Think of the Post Byte as 8 bits and set them as shown in the chart below. Bits 5&6 specify the register and should be set as shown below the chart. - means that contents of Bit is unimportant.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | ADDRESSING MODE |
|---|---|---|---|---|---|---|---|---|
| 1 | | | 0 | 0 | 0 | 0 | 0 | Autoincrement by 1 |
| 1 | | | 1 | 0 | 0 | 0 | 1 | Autoincrement by 2 |
| 1 | | | 0 | 0 | 0 | 1 | 0 | Autodecrement by 1 |
| 1 | | | 1 | 0 | 0 | 1 | 1 | Autodecrement by 2 |
| 1 | | | 1 | 0 | 1 | 0 | 1 | Accumulator B offset |
| 1 | | | 1 | 0 | 1 | 1 | 0 | Accumulator A offset |
| 1 | | | 1 | 1 | 0 | 0 | 0 | 8 Bit offset |
| 1 | | | 1 | 1 | 0 | 0 | 1 | 16 Bit offset |
| 1 | | | 1 | 1 | 0 | 1 | 1 | Accumulator D offset |
| 1 | - | - | 1 | 1 | 1 | 0 | 1 | PC 16 BIT offset |
| 1 | - | - | 1 | 1 | 1 | 1 | 1 | Extended Direct |

BIT NO. above bits 7–0.

| 6 5 | Set Bits for Register |
|---|---|
| 0 0 | X register |
| 0 1 | Y register |
| 1 0 | U register |
| 1 1 | S register |

Here are some common Post Bytes in decimal

| | |
|---|---|
| Autoincrement X by 1 | 128 |
| Autoincrement Y by 1 | 160 |
| Autodecrement X by 1 | 130 |
| Autodecrement Y by 1 | 162 |
| | |
| Autoincrement X by 2 | 129 |
| Autoincrement Y by 2 | 161 |
| Autodecrement X by 2 | 131 |
| Autodecrement Y by 2 | 163 |

# APPENDIX C

## SOME USEFUL ROM ROUTINES

| | |
|---|---|
| 48101 | Check Keyboard & place result in A |
| 46004 | Restart Basic |
| 48466 | Checks the Joystick |
| 48691 | Turn on cassette relay |
| 48604 | Turn off cassette relay |
| 48744 | Prepare cassette for writing |
| 48615 | Prepare for data input |
| 48658 | Put out contents of A to cassette |
| 48557 | Input 1 character from tape to A |
| 48299 | Write contents of A to screen |
| 48053 | Blinks Cursor |