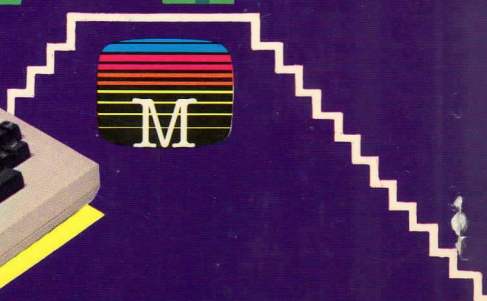


# **DRAGON**

***machine language for  
the absolute beginner***



**JOHN VANDER REYDEN**

Published in the United Kingdom by  
Melbourne House (Publishers) Ltd.,  
Melbourne House,  
Church Yard,  
Ting, Haverhill, HPS 8LU,  
ISBN 0 581 61 130 8

# ***DRAGON***

## ***machine language for the absolute beginner***

Published in the United States of America by  
Melbourne House Publishers Inc.,  
347 Passaic Drive,  
Haverhill, MA 01831

This book was edited by John Vander Reyden  
with contributions from Denver Jones and Craig Hill et al.

Copyright © 1985 Benn Cohen

All rights reserved. This book is copyright. All part of this book may be  
copied or stored by any means whatsoever without permission or  
electronic except for private use in the Copyright Act.  
All enquiries should be referred to

Printed in Great Britain by Colson

**JOHN VANDER REYDEN**



**MELBOURNE HOUSE**

Published in the United Kingdom by:  
Melbourne House (Publishers) Ltd.,  
Melbourne House,  
Church Yard,  
Tring, Hertfordshire HP23 5LU,  
ISBN 0 86161 130 6

Published in Australia by:  
Melbourne House (Australia) Pty. Ltd.,  
Suite 4, 75 Palmerston Crescent,  
South Melbourne, Victoria, 3205,  
National Library of Australia Card Number and  
ISBN 0 86759 125 0

Published in the United States of America by:  
Melbourne House Software Inc.,  
347 Reedwood Drive,  
Nashville TN 37217.

This book was edited by John Vander Reyden,  
with contributions from Denver Jeans and Craig McFarlane

Copyright © 1983 Beam Software

All rights reserved. This book is copyright. No part of this book may be copied or stored by any means whatsoever whether mechanical or electronic, except for private or study use as defined in the Copyright Act. All enquiries should be addressed to the publishers.

Printed in Hong Kong by Colorcraft Ltd.  
1st Edition

# Contents

<b>1. The Beginning</b> .....	1
<b>2. What is a Micro-computer?</b> .....	3
<b>3. Computers and Numbers</b> .....	5
Binary .....	5
Negative Numbers .....	7
Exponential Numbers .....	8
Hexadecimal Numbers .....	10
Binary Coded Decimal .....	11
Decimal-Hexadecimal-Binary-Conversion Program .....	12
<b>4. What is Machine Language?</b> .....	14
<b>5. What is Assembly Language?</b> .....	17
<b>6. The Dragon</b> .....	20
<b>7. The 6809</b> .....	23
Registers .....	23
Modes .....	25
<b>8. Easy</b> .....	31
<b>9. Handy</b> .....	35
<b>10. Let's Get Logical</b> .....	39
<b>11. Condition Codes</b> .....	42
Decisions, Decisions .....	46
Loops .....	47
<b>12. Stacks and Subroutines</b> .....	52
<b>13. The 6809 Instruction Set</b> .....	60
<b>14. Demonstration Programs</b> .....	127
Introduction .....	127
a) The No OPeration Instruction .....	129
b) The Complete Byte and Register Handlers .....	129

c) The Arithmetic Instructions .....	133
d) The Logical Instructions .....	139
e) Comparisons .....	141
f) The Branch and Jump Instructions .....	143
g) The Rotate Instructions .....	145
h) The Stack Handling Instructions .....	146
i) The Interrupt Instructions .....	146
<b>15. Programming Your Dragon .....</b>	<b>147</b>
Planning Your Machine Language Programs .....	147
Entering and Running Machine Language Programs .....	151
Monitor Program .....	155
<b>16. Sample Programs .....</b>	<b>156</b>
Introduction .....	156
The PIA (Peripheral Interface Adaptor) .....	156
Screen Memory .....	159
The Hardware .....	159
The Use of the Direct Page .....	160
Program: PIA Keys .....	162
Program: Score .....	167
Program: Explode .....	178
Program: Music .....	191
Program: Demo .....	198
<b>Appendix A: Colour Set Table .....</b>	<b>221</b>
<b>Appendix B: Graphics Modes .....</b>	<b>222</b>
<b>Appendix C: Handy Memory Locations in the Dragon .....</b>	<b>237</b>
<b>Appendix D: Handy ROM Routines .....</b>	<b>239</b>
<b>Appendix E: ASCII Codes for Keys .....</b>	<b>242</b>
<b>Appendix F: Character Codes .....</b>	<b>244</b>
<b>Appendix G: Base Conversions .....</b>	<b>245</b>
<b>Appendix H: 6809 Instruction Set Summary .....</b>	<b>249</b>





## CHAPTER 1

# The Beginning

This book is designed as an introduction to machine and assembly language programming on the 6809 chip and specifically on your DRAGON.

Obviously you must have some idea of what machine language is or you wouldn't have this book. It would be fair to assume, however, that your knowledge of machine code extends no further than knowing that it is a computer language, that it is a list of strange numbers and letters and that it is supposed to be far superior to BASIC. Perhaps you aren't even aware of this or aware that there is a difference between assembly and machine language, nor indeed how they differ from BASIC.

What ever you do or don't know about machine language, don't worry, and don't be frightened by the jargon. We have made no assumptions about your familiarity with machine language programming and we will explain all the jargon as well. This book is intended to be used by the absolute beginner and was written specifically for the absolute beginner.

To start we are going to explain exactly what a micro-computer is and how it works. Of course this is only going to be brief, as to understand everything would take years of study and a small library of books.

Part of the mystique of computers is the funny numbers that they use. These will be revealed, they are really not that



hard to learn, just that computer people like to keep the public at large "in the dark" (it makes them feel more important).

After learning what a computer is and how it does things you are ready for the wonderful world of 'machine language'. Again not as hard as computer people would have you believe.

Finally a few sample programs to give you the idea of how it is done and the rest is up to you.

Happy programming!

CHAPTER 1

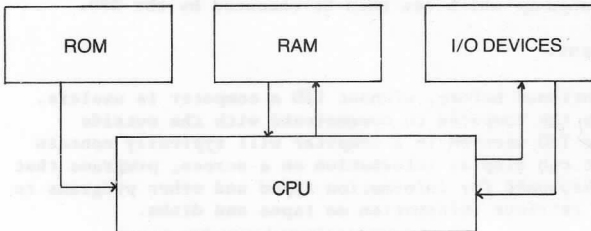
# The Beginning

## CHAPTER 2

# What is a micro-computer?

In the next few sections we will describe in general terms what a micro (computer) is and later on the DRAGON in particular.

A micro is a very simple machine. Functionally it has Just three sections, the CPU (Central Processor Unit), it's memory and I/O (Input/Output) as shown in diagram 1.



Diag. 1

### The CPU.

The CPU is the heart of any computer (in a micro it is a single chip) although without both of the other two sections it is useless. Think of your brain as if it was completely cut off from all your senses (sight, hearing, etc..), it still has memory but what can it do? And without memory it would be just as bad. It would be able to see, hear, etc. but it wouldn't know what it was seeing or hearing.

The CPU can do only simple things, i.e. it can add and subtract, compare numbers, tell the memory what to remember and retrieve numbers when they are needed.

### The Memory.

Inside a micro there are two types of memory (excluding outside memory i.e. tapes, disks, etc.) and they are RAM and ROM which sound impressive but just stand for Random Access Memory and Read Only Memory.

### RAM.

You can read and write to RAM but it forgets things fairly quickly unless it is continuously reminded (the SAM - a chip described later - does this). RAM is the memory that is used for holding onto information that changes, variables etc, as well as your programs.

### ROM

On the other hand ROM cannot be changed and it never forgets anything even when the power is off (unless it blows up). This is where the "Operating System" is stored. The OS is in reality a huge machine language program that makes the computer easier for you to use. In the popular home type computer it contains a BASIC interpreter which simply takes the BASIC commands that you type in and converts them to machine language which can then be executed by the CPU.

### Input/Output.

As was mentioned before, without I/O a computer is useless. I/O allows the computer to communicate with the outside world. The I/O section in a computer will typically contain chips that can display information on a screen, programs that scan the keyboard for information typed and other programs to store and retrieve information on tapes and disks.

All these sections will be described in more detail when we get to look at the DRAGON.

## CHAPTER 3

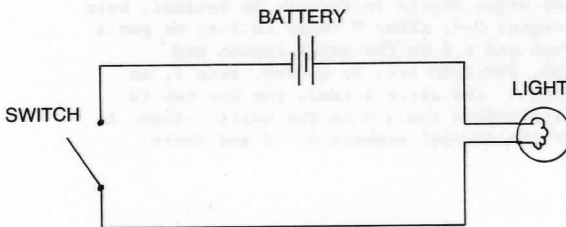
# Computers and Numbers

### Binary

Now that we have described what a computer is (you probably knew all that anyway), it is time to get into some "heavy stuff".

Everyone knows that computers can do arithmetic, but few realize how. Those humans who can still manage arithmetic, do it in decimal. This means we count the numbers in powers of 10, i.e. units, tens, hundreds, thousands, etc.. Computers on the other hand work in powers of two, or base two, or binary. If you haven't caught on as to why computers use binary, hold on.

Think of a simple electric circuit consisting of a switch, battery, and light bulb as in diagram 2.



Diag. 2

When the switch is open, the light bulb will be off. But if we close the switch, the light will shine. So we have two states for the bulb, ON or OFF. Or, if you want to put it in numbers, we have a 0 for off, and a 1 for on. Getting the idea? Two states? Binary? That off or on can be termed a BIT or Binary digit of either 0 or 1 respectively. The reason I am using the switch and light bulb for an analogy should be getting clearer now. Yes, that's right, just about any computer can be thought of as a glorified collection of switches (light bulbs are optional).

To understand how binary works, it's a good idea to recap our decimal first. Say we have on our hands the decimal number 2,153. Let's have a look at just what the number means:

T	H	T	U
2	1	5	3

Basically, it means 2 Thousands, 1 Hundred, 5 Tens and 3 Units. Or if we express it in powers of 10:

2	1	5	3
3	2	1	0
10	10	10	10

3            2            1            0

It would be:  $2*10^3$ ,  $1*10^2$ ,  $5*10^1$ , and  $3*10^0$ . If Base 10 works this way then we would expect numbers represented in Base 2 to be grouped in lots of the following:

7	6	5	4	3	2	1	0
2	2	2	2	2	2	2	2

or

128	64	32	16	8	4	2	1
-----	----	----	----	---	---	---	---

To give an example, let's look at the decimal number 137:

$$137 = (1*128) + (0*64) + (0*32) + (0*16) + (1*8) + (0*4) + (0*2) + (1*1)$$

or, in binary

$$= 10001001$$

Note that the only digits used are 0's and 1's. This is because there are no other digits in binary. In decimal, base 10, there are the digits 0-9, after 9 comes 10 i.e. we put a 1 in the ten's column and a 0 in the units column and similarly for 99/100, 999/1000 etc. In binary, base 2, we have the digits 0 and 1 and after 1 comes two but two is really a 1 in the 2's column and a 0 in the units column. As an example here are the decimal numbers 0 - 8 and their binary equivalent

0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

Most of the micro's around these days are 8-bit, which means that the numbers they work with are 8-bits long, or to give it another name one byte long. If you are fast you have already worked out that the largest decimal number that can be represented by one byte is 255. For the slower of you it is calculated as follows; to get the largest number, all the digits would have to be the largest they can be (1's). This gives us 1111 1111 and when that is converted ( $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$ ) gives us 255.

To make conversion easier and allow you to play around with different numbers and see how they look there is a program at the end of the chapter which converts either decimal, binary or hexadecimal numbers into the other two and displays all three (don't panic about hexadecimal numbers we will be covering them soon).

So far binary looks reasonable for positive whole numbers up to decimal 255 but what about numbers bigger than 255 or negative numbers or fractions?

## Negative Numbers

Pretend that we want to see how the computer would store the number -73. First we have to work out what 73 is in binary:

$$73 = 01001001 \quad (\text{using the method above})$$

Now we "complement" the binary form of 73. This is just a matter of looking at each bit in the number, and where we see a 0 we put a 1, or vice versa. Therefore we should have:

$$10110110$$

And now for the finishing touch, which is just to add 1 to the result to give:

$$10110111$$

So the method involved in negating a number is simple, complement the number, then add 1 to it. However, by doing this we produce some limitations on the size of a positive or negative number. This comes about because of the MSB (Most Significant Bit, or the left-hand most bit which, in fact, has the highest value). When we are using signed numbers, we can tell if a number is negative or positive because, if it

is negative, the MSB will be set. Positive numbers have a MSB clear or 0. From this we can see that the largest positive number we can have using eight bits and 2's complement is:

$$01111111 = 127$$

and when we add 1 to this the most negative number we can have is:

$$10000000 = -128$$

The reason 2's complement was chosen to represent negative numbers is that it allows us to add positive and negative numbers together and come out with the right result. For example,  $-5 + 3$ :

$$\begin{array}{r} 5 \text{ dec} = 0000\ 0101 \text{ bin} \\ -5 \text{ dec} = 1111\ 1011 \text{ bin} \\ + 3 \text{ dec} = 0000\ 0011 \text{ bin} \\ \hline = -2 \text{ dec} = 1111\ 1110 \text{ bin} \end{array}$$

To convert a negative 2's complement number to decimal, simply reverse the procedure that converts negative decimal to 2's complement. Therefore you would subtract 1 from the number and complement the binary result (make all 1's, 0's etc.).

You may be wondering how you would tell the difference between an unsigned number so large that its MSB is set and a truly negative number. The answer is, remember what type of number it's supposed to be. This is not really a disadvantage since you, as the programmer, should remember whether the number is supposed to be signed, i.e.  $-128$  to  $+127$ , or unsigned, i.e.  $0$  to  $255$ .

So much for negative numbers, now for the hard stuff - large numbers and fractions.

The CPU in the DRAGON (6809) can also handle numbers two bytes long without much trouble. This allows numbers up to  $65,535$ . If you don't know where this came from, try adding up this:

$$\begin{array}{cccccccc} 15 & 14 & 13 & & & & 1 & 0 \\ 2 & + 2 & + 2 & + \dots & + 2 & + 2 & = 65,535 \text{ or} \\ +322767/-32768 & \text{when using 2's complement for signed numbers.} \end{array}$$

But what about fractions and numbers even bigger than  $65,535$ ?

## Exponential Numbers

There is another way to represent numbers in binary and that is exponentially. This is sometimes known as scientific notation when used with decimal numbers. Basically there are

two parts to a number; the exponent and the mantissa. The mantissa is a number (usually 4 bytes long), which when multiplied by, 2 raised to the power contained in the exponent (usually 1 byte), gives the decimal number being stored.

To convert a decimal number, X, to this type of representation follow the procedure given below.

i) If  $X = 0$  then all the bits in both the exponent and the mantissa are 0s

ii) Convert decimal to binary - leave the decimal point in it's place, this now becomes the binary point. To convert the numbers to the right of the decimal point you use much the same procedure as when converting whole numbers except that the columns, instead of representing 1, 2, 4, 8, 16, etc., represent  $1/2$ ,  $1/4$ ,  $1/8$ ,  $1/16$ , etc. (from the binary point going right). The table below gives the first 8 fractions and their decimal equivalent:

$1/2$	0.5
$1/4$	0.25
$1/8$	0.125
$1/16$	0.0625
$1/32$	0.03125
$1/64$	0.015625
$1/128$	0.0078125
$1/256$	0.00390625

From these numbers it is quite simple to convert decimal fractions to binary fractions as below:

- If the decimal fraction is larger than or the same as 0.5, put a 1 in the  $1/2$  column and subtract 0.5 from the original decimal fraction, otherwise put a 0 in the  $1/2$  column.
- Repeat a) using 0.125 and the  $1/4$  column then 0.0625 and the  $1/8$  column etc.
- Keep doing b) until you have enough significant digits or the original decimal fraction becomes zero.

iii) Exponent:

- if there are any digits to the left of the binary point (the old decimal point) then the exponent equals the number of digits to the left.
- if the first digit to the right of the binary point is a 1 then the exponent is zero.
- otherwise the exponent equals the complement of the number of 0s, going left to right from the binary point, up to the first 1.

iv) Mantissa:

- remove the binary point and all leading 0s and add 0s until you have the correct number of digits to fill the available bytes (usually 32).
- if the original number was positive, change the first 1 to a 0.

v) Make into groups of eight and convert to hex.

For a step by step example we will convert decimal 12.375 and 0.15625 to exponential format.



X = 12.375

i) X is not 0 then do nothing.

ii) convert to binary. 12.375 - 1010.011

iii) Exponent:

- the number of digits to the left of the binary point is 4 therefore the exponent is \$04

iv) Mantissa:

- 10100110 00000000 00000000 00000000

- 00100110 00000000 00000000 00000000

v) \$26 \$00 \$00 \$00 \$04

Don't panic about the "\$" sign in front of the numbers, It simply denotes the number as being hexadecimal (explained in the next section).

X=0.15625

i) X is not 0 then do nothing.

ii) convert to binary. 0.15625 - 0.00101

iii) Exponent:

- the number of digits to the right of the binary point, up to the first 1, is 2 therefore the exponent is the complement of 2 = \$FE

iv) Mantissa:

- 10100000 00000000 00000000 00000000

- 00100000 00000000 00000000 00000000

v) \$20 \$00 \$00 \$00 \$FE

To convert from this representation into decimal simply raise 2 to the power contained in the exponent and multiply by the mantissa.

Using this representation decimal numbers, both positive and negative, from less than 1 thousandth to over 1,000 million can be stored with an accuracy of 9 decimal digits.

## Hexadecimal Numbers

By now you must be seeing 1's and 0's in your sleep. The fact is, binary is a real pain to use. Nobody bothers with representing numbers in binary unless he really has to. But by the same token, decimal just doesn't seem to fit in binary's place. To overcome the difficulty of trying to read binary whilst still keeping a similar form, computer people use "hexadecimal". What is hexadecimal?! Don't panic, hexadecimal is only a term we use for base 16. Why base 16? It works like this. Four bits can contain sixteen different values, 0 - 15. A hexadecimal digit can have sixteen values too, 0 - 15. A single hexadecimal digit can be used to stand for four binary digits:

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

What's this A,B,C,D,E and F business! Why not just use 10, 11, 12, 13, 14 and 15? We don't use them because they are numbers, not digits. What we need are digits that go higher than 9 and up to 15. When the Arabs invented digits, they didn't have much foresight and neglected any possibility of needing such digits, four thousand years later. So, we have to invent some new ones, hence the A, B, C, D, E and F.

Take the decimal number 2,153 again. In binary it would look like:

100001101001

Fearsome isn't it! On the other hand, the hexadecimal representation is just:

869

Much nicer, but it doesn't look any different from a decimal number. To stop the confusion between the two we put a dollar sign "\$" in front of any hexadecimal number we use, just to let us know that it is hexadecimal:

\$869

## Binary Coded Decimal

Just to make matters more confusing there is one more way to represent numbers in machine language and that is Binary Coded Decimal (BCD). This form allows decimal numbers in binary.

First, you should know what a "nybble" is. If a bit is one binary digit and a byte is eight bits, a nybble could be expected to lie between, and it does. A nybble is four bits grouped together.

Back to BCD. To encode each of the digits 0 to 9, only one nybble is needed and six of the possible codes available in a nybble are ignored. Since one nybble is needed for each digit, two digits can be held in the same byte. For example:

0000 0000 is BCD for decimal 00.  
 1001 0101 is BCD for decimal 95.  
 1010 1100 is an invalid number in BCD.

Binary nybbles which represent decimal numbers greater than 9 are invalid.

This convention for representing numbers leads to potential numbers in addition and subtraction. For example:

BCD 08	0000 1000
+ BCD 03	0000 0011
= BCD 11	0000 1011

Notice that the result of this operation is an invalid BCD number. To compensate for this the result must be adjusted. The adjustment required is to add 6 to the result if the result is greater than 9. In the above example:

```

      BCD 08      0000 1000
+   BCD 03      0000 0011
-----
                        0000 1011
      result greater than 9? Yes; add 6
                        + 0000 0110
-----
=   BCD 11      0001 0001

```

The CPU in the DRAGON (6809) has special instructions for handling BCD numbers and these will be shown to you later.

You will be unlikely to use BCD in your programming but it is there and can be used. The only time it is valuable is when highly accurate, whole number arithmetic is needed as exponential representation sometimes leads to rounding errors.

## Decimal-Hexadecimal-Binary-Conversion Program

This program allows you to enter either a decimal, hexadecimal or binary number and then displays that number in the above forms with the decimal both signed and unsigned.

After keying in the program and saving it, you are ready to run it. The program prompts you for a number, you then enter your number as follows; decimal - between -128 and 255, hexadecimal - between \$00 and \$FF remembering to include the \$ as the first character, and binary - up to 8 digits (0 or 1) preceded by a %.

After the number has been converted and displayed you are again prompted for a number, you may continue to enter different numbers as long as you like. When you wish to stop the program, simply key ENTER directly after the prompt.

Note that if you want to do two byte conversions between decimal and hex, for addresses, there are standard BASIC functions available directly from your Dragons keyboard. For decimal to hex, use "?HEX\$('decimal number')", and for hex to decimal use "?&H'hex number'".

```

10 LINE INPUT "ENTER NUMBER : ";A$
20 IF A$="" THEN END
30 PRINT
40 IF LEFT$(A$,1)="#" THEN 90
50 IF LEFT$(A$,1)="$" THEN 170
60 ' DECIMAL
70 A=VAL(A$)
80 GOTO 200
90 ' BINARY
100 A=0
110 A$=MID$(A$,2)
120 A=2*A

```

```

130 IF LEFT$(A$,1)="1" THEN A=A+1 : GOTO 150
140 IF LEFT$(A$,1)<>"0" THEN 380
150 A$=MID$(A$,2)
160 IF A$="" THEN 200 ELSE 120
170 ' HEXADECIMAL
180 IF LEN(A$)>3 THEN 380
190 A=VAL("&H"+MID$(A$,2))
200 ' A=VALUE
210 IF A<-128 THEN 380
220 IF A<0 THEN A=A+256
230 IF A>255 THEN 380
240 B=A
250 A$=""
260 FOR I=1 TO 8
270 IF B/2-INT(B/2)=0 THEN A$="0"+A$ ELSE A$="1"+A$
280 B=INT(B/2)
290 NEXT I
300 PRINT "BINARY :", "%"+A$
310 PRINT "HEX :", "$"+HEX$(A)
320 PRINT "DECIMAL"
330 PRINT " UNSIGNED :", A
340 IF A>127 THEN A=A-256
350 PRINT " SIGNED :", A
360 PRINT
370 GOTO 10
380 PRINT"* INVALID INPUT"
390 PRINT
400 GOTO 10

```

## CHAPTER 4

# What is Machine Language?

By now you might have forgotten that this is what we intended to learn about in the first place so it is a pretty good thing to define machine language before we go much further. My dictionary says, "Machine Language: the coding system adopted in the design of a computer to represent the instruction repertoire of the computer". If that sentence sent your head spinning, fear not for we had the same problem.

You have already seen how numbers are handled by computers. And machine language is only a lot of numbers like we discussed before. The difference is that whereas we were thinking of these numbers as just numbers, machine language treats them as more than such. A particular number when used in machine language, will cause the CPU to perform a particular activity or instruction.

For example: \$8B, decimal 139 or binary 1000 1011 would cause the CPU to add two numbers together. Quite a long way from something like

$$A = A + 1$$

Nonetheless, this is what machine language is all about. The name says it all! It is language for machines. Each manufacturer of the different CPUs has designed a different language for its product.

At this stage you may be asking yourself - if this is what machine language is all about, why bother? Why not accept the benefits of someone else's work which allows me to program the computer in a language that I can easily understand, such as BASIC?

The reason you should bother is because of the benefits of machine language, these are:

- \* FASTER EXECUTION OF THE PROGRAM
- \* MORE EFFICIENT USE OF MEMORY
- \* SHORTER PROGRAMS (in memory)
- \* FREEDOM FROM THE OPERATING SYSTEM

All of the above benefits are a direct result of programming in a language that the CPU can understand without having to have it translated first. When you program in BASIC, the operating system is a machine language program that is being run by the machine. The program could be described like this:

Next Look at the next instruction  
Translate it into a series of  
machine language instructions  
Perform each instruction  
Store the result if required  
Goto Next

Programming in BASIC can be up to 60 times slower than a program written directly in machine language! This is because translation takes time, and also the resulting machine language instructions generated are usually less efficient. Similarly it is usually faster to drive yourself than to take public transport; you can take shortcuts you know instead of following the public transport route which needs to cater for the general public's needs.

However, we would also be among the first to admit that programming in machine language does have drawbacks.

The main disadvantages of machine language are:

- \* PROGRAMS ARE MORE DIFFICULT TO READ AND DEBUG
- \* IMPOSSIBLE TO ADAPT TO OTHER COMPUTERS
- \* LONGER PROGRAMS (in instructions)
- \* ARITHMETIC CALCULATIONS DIFFICULT

We do not advocate that you write every program you want in machine language instead of BASIC. You need to look at the programming task in question and determine which language is best suited to development of that program. A program used for calculating averages is a simple one when written in BASIC, but a machine language version would take a lot more effort. On the other hand, you could die of boredom watching an arcade game written in BASIC creep along. Guidelines for choosing between the two are:

a) Memory Requirements - BASIC is usually a permanent resident in ROM, in home micros, and as such only uses RAM to store your BASIC program and its variables. Routines to do things like arithmetic, etc. are all done in the BASIC ROM. To do the equivalent in machine language, you would have to have your own routines in RAM and this uses more of your highly valuable RAM.

b) Time Available for Development - Even if you can develop a quicker, shorter program in machine language than the BASIC equivalent, it still may not be worthwhile. Putting it bluntly, time is money and is often the overriding factor in such decisions.

c) Speed Requirements - If you simply must have speed then machine language is the way to go. However, if it really isn't important then BASIC would probably be better.

d) Number Crunching - If your application involves a lot of arithmetic, then the language best suited to handling numbers should be used. This goes especially for situations involving real or floating point numbers. For this type of programming BASIC is better.

## CHAPTER 5

# What is Assembly Language?

Quite obviously if machine language could only be represented by numbers, very few people would want to write programs in machine language.

After all, who would make sense of a program which looked like:

```
0 0 1 0 0 0 0 1   or   $21
0 0 0 0 0 0 0 0   or   $00
0 1 0 0 0 0 0 0   or   $40
etc.
```

Fortunately, we can invent a series of names for each of these numbers. Assembly language is just such a representation of machine language, enabling it to be read by humans.

The main difference between assembly language and machine language is that assembly language is one level higher than machine language. It is more easily read by humans than machine language, but on the other hand, computers can't read assembly language.

It is not an adaptation of machine language such as BASIC. For each assembly language instruction there is an identical (in function) machine language instruction, and vice versa; i.e. there is a ONE-TO-ONE relationship between them. We can therefore say that assembly language is equivalent to machine language. Assembly language makes use of mnemonics (or



abbreviations) to enhance readability. For example, at this stage, the instruction

INCA

may not mean much to you but at least you can read it. If you were told "INC" is a standard mnemonic (say "mnemonic") for INCrement and that A is a variable, then by simply looking at the instruction you can get a feel for what is happening.

The same instruction in machine language is

0100 1100

Now obviously you can also "read" that instruction in the sense that you can read the number, but it doesn't mean much unless you have a table to look up or, your brain is capable of functioning like a computer.

This is only scratching the surface of assembly language. You can do a lot more than write mnemonics instead of numbers for instructions. A line of an assembly language program can be divided into sections like this:

LABEL MNEMONIC OPERAND COMMENTS

You already know what a mnemonic is but what is all the other stuff? We start with the easiest first.

COMMENTS - As you have probably already guessed, a comment is a few words which don't effect the actual program but, like a REMark in BASIC, is there to remind you, or tell other people that may look at your program, exactly what you are doing in this section of the program.

OPERANDS - We said before that there was an instruction that added two numbers together but we didn't say what these numbers were. This is what the operand does. It can tell the CPU which numbers to use or, in other cases the mnemonic tells the CPU what instruction to execute and the operand tells the CPU what to use that instruction with.

LABELS - These are a godsend. In BASIC if you have ever moved a subroutine then you will have had to go through all your code and find the places that called that subroutine and change the line numbers on the GOSUB. Labels let you give a name to a line and then all you have to do is the equivalent of a GOSUB or GOTO to that name no matter where it is.

Labels also allow you to give names to other things such as complete programs, constants, variables, etc. This is very useful in assembly language programs as you will see later on.

Assembly language can be converted directly into machine code by a program or by you. Such a program is called an assembler. You can see that this is a program which performs the rather boring task of translating your assembly language program into a sequence of machine language instructions that the CPU will understand i.e. into binary numbers.

The alternative to an assembler program is for you to do the translation of the assembly language into machine code by hand, using the tables provided in this book.

It's hard, it's frustrating at first, it's inconvenient but it's wonderful practice and gives you great insight into the way the CPU works. We recommend that you try writing short machine language programs in this way - i.e. writing them in assembly language and then translating them into machine language by hand - before buying an assembler.

## CHAPTER 6

# The Dragon

Up to this stage we have been talking in general terms, all the things described above are applicable to most home computers but now we will talk specifically about your computer, the DRAGON.

If you dare, take the cover off your DRAGON by undoing the screws underneath the warranty stickers on your computer and pulling the top off.

**WARNING!:** Doing this will void your warranty so we don't advise you to attempt this if you wish to retain a valid warranty!

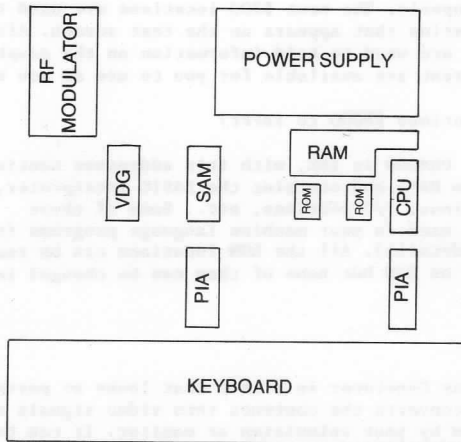
You can now see the circuit board in all its glory and all the chips that make the DRAGON work. To give names to the important chips and for those of you who don't want to take the risk of opening up your computer, see diagram 3.

### The CPU

You already know what the CPU does in general terms so we will not discuss it at length here except to say that it is the Motorola designed 6809, one of the most powerful 8-bit CPUs available today.

### The MEMORY

You also know roughly what the RAM and ROM are but we will be more specific here.



Diag. 3

RAM (memory locations \$0000 to \$7FFF)

The DRAGON that you own comes with 32K RAM (A "K" is explained later). This means it has 32,768 different memory locations (or addresses, if you like). We can store one byte of information at each address and we use two bytes when referring to the address (this allows us to address each location in memory). At this stage we will describe some things related to memory, a "K" and a "page".

i) A "K" (as you might have guessed) is a number around about one thousand. In fact, it is equal to 1024. Remember that computers work in binary and not decimal and that 1024 works out to be a nice simple hexadecimal number, \$400. It is used because it is the binary equivalent of the metric K (Kilo or 1000), if you like. It is sort of a natural number in hex. just as a thousand is a natural number in decimal.

ii) A "page" is 256 or \$100 memory locations (e.g. \$00 to \$FF, \$100 to \$1FF, etc.) sharing the same upper byte of their addresses. For example, \$B300 to \$B3FF is one page and when all the addresses are split into two bytes, the first byte is always \$B3 (for this page). The second byte defines which location in that page is to be addressed. As you might expect we can assign page numbers to these blocks of memory. In the above example, the page number is \$B3.

The first \$400 locations are used by BASIC and the operating system so these are not really available for use if you want BASIC to run properly. The next \$200 locations are used to hold the information that appears on the text screen. After that, some more are used to hold information on the graphics screen and the rest are available for you to use as you wish.

ROM (memory locations \$8000 to \$BFFF)

The ROM in your DRAGON is 16K, with it's addresses continuing directly on from RAM, and contains the BASIC interpreter, arithmetic routines, I/O routines, etc.. Some of these routines may be used in your machine language programs (see Appendix D for details). All the ROM locations can be read in the same way as RAM but none of them can be changed in any way.

The VDG

The Video Display Generator is a chip that looks at parts of the memory and converts the contents into video signals that can be displayed by your television or monitor. It can be used to display simple text or the most complex of graphic designs.

The SAM

This chip, as mentioned earlier in the book, continuously reminds the RAM of its contents. It also controls all access to memory by the CPU and VDG. The CPU when it wants to access memory asks the SAM. The SAM then produces that particular byte for the CPU. The SAM is also continuously feeding the VDG with the contents of the relevant parts of memory so that it can produce video signals.

The PIAs

These two chips, Peripheral Interface Adapters, do all the I/O of the computer. They allow the CPU to talk to your peripherals, such as printer, cassette, etc, and also allow the CPU to tell the SAM and VDG what mode they are to run in.

Each of these chips are described in more detail in the sample programs chapter along with programs showing how to use them.

## CHAPTER 7

# The 6809

## Registers

The CPU uses a number of things in its dealings with ROM, RAM and I/O devices. Some of the main parts which you will use are registers. These registers are just shorthand number storage places that the CPU uses to execute programs.

Think of when you do arithmetic by hand. You would have some numbers and an operation on a piece of paper, say  $4 + 6$ . You look at this piece of paper and think to yourself, " $4 + 6$  that's 10". You then write the result of 10 on the piece of paper.

The CPU does something very similar when the time comes for it to do arithmetic like that. Firstly, it gets the numbers (data) and the operations (program) from its "piece of paper" (memory). Then it thinks to itself like you did. To do that, however, it needs somewhere to keep those numbers and operations while it thinks about them. It doesn't use memory, that's much too slow. It would be like us writing every single thought on paper! So it uses registers and they all have names:

A B D CC DP X Y U S PC

These registers come in various sizes and can be used for various purposes. Let's take a closer look:

A - is Accumulator A. This is an 8-bit wide register. Accumulators are primarily used for arithmetic and logical functions on the numbers they contain. The 6809 in fact has two 8-bit accumulators, the A and B registers. These two accumulators are almost identical, the only difference between them comes about because a few instructions treat them differently. We'll talk about the differences later.

B - is Accumulator B. As mentioned above, the B register is a near identical twin of the A register.

D - is a con-trick! There is actually no special D register. The D register is supposedly a 16-bit wide accumulator, but in reality is just the A and B accumulators joined together! The A register holds the high byte and the B register keeps the low one. Throw in a few special 16-bit arithmetic operations and hey presto! The D register. This feature makes 16-bit arithmetic a real snap for 6809 programmers.

CC - The Condition Code register is just a collection of flags living in an 8-bit register. Flags are just like our light bulb we used earlier on, i.e. a bit that is either ON or OFF (SET or RESET). These flags and statuses reflect the state the 6809 is in at any given time. These include flags used in arithmetic and logical operations, as well as several others. The CC register has a chapter all of it's own later on.

X - Is Index register X. Index registers are 16-bit registers which are normally used as "pointers". A pointer is just a 16-bit number that contains the address of a particular location in memory i.e. it "points" to an address.

Y - Is Index register Y. The Y register will do almost anything that the X register can do. However, the X register is normally preferred because the 6809 can use it a little more quickly and in fewer bytes, in a program, than it can the Y register, however we do use it when the X register is being used for something else and we need an Index register.

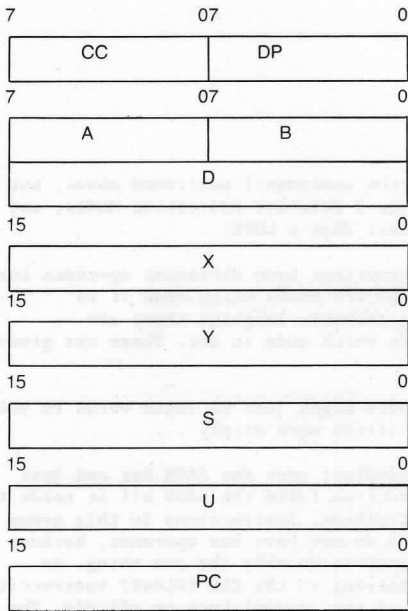
S - is the Stack Pointer. This is a very special 16-bit register which is responsible for keeping what's known as the 6809's "stack" in order. Stacks will be explained later, but for now I will stress that this stack and its order are essential for the 6809 to go about executing programs properly.

U - is the User Stack Pointer. This is also a very special and versatile 16-bit register. Not only can it be used as a stack pointer without effecting the 6809's program execution, like the S register, but it can also be used as a 16-bit index register like the X and Y registers!

DP - is the Direct Page register. In the previous chapter I explained what a page was. This register allows a short hand way of referring to locations within a page of memory. The DP register contains a page number specifying a page of memory that we wish to refer to often. This is used in the direct addressing mode. (More about addressing modes later)

PC - is the Program Counter. This is the 16-bit register which always points to the next instruction the 6809 has to execute. This is the register which lets the 6809 run machine language programs and as you can probably guess, is also essential for the 6809 to serve any useful purpose.

To make it easier to remember all these different registers diagram 4 set them out in a logical manner.



Diag. 4

## Modes

Modes are the different ways we can do the same thing. For instance, to get to work, we can: drive, catch a train, walk, etc.. Whichever mode we choose, we still get to work. However, some ways are quicker, or more convenient than others, no matter which mode you eventually choose, it's



still nice to have as many options as you can. An addressing mode tells the CPU how to get the operands (numbers) it needs to execute a particular instruction, for example Immediate Mode tells the CPU that the operand it needs is included in the program immediately after the instruction (like a constant in BASIC) where as Extended Mode means that the operand is in another section of memory altogether (more like a variable in BASIC).

This is another feature which makes the 6809 such a powerful CPU. There are numerous Addressing modes (quite a few more than other 8-bit CPU's)

Let's get into them ...

The 6809 has a total of 7 basic types of Addressing Modes:

- Inherent
- Register
- Immediate
- Relative
- Direct
- Extended
- Indexed

This may not sound like the numerous I mentioned above, but when we see that there are 2 Relative Addressing Modes, and 27 separate Indexing Modes! That's LOTS!

In machine code the instructions have different op-codes for the different modes so the CPU knows which mode it is supposed to be using. In assembly language there are different ways to specify which mode to use. These are given below.

Now so far, all these modes might just be vague words to you, so let's explore them a little more deeply ...

**Inherent** - This is the simplest mode the 6809 has and just the op-code of the instruction tells the 6809 all it needs to know to perform the instruction. Instructions in this group are very specialised, and do not have any operands, because the instruction is designed to do only the one thing. An example of inherent addressing is the CLR (CLear) instruction which simply clears one of the accumulators to all 0's. The mnemonics are; CLRA to clear the A accumulator and CLRB for the B accumulator and the op-code for each is different (\$4F and \$5F respectively) so that the CPU knows what to do and which accumulator to do it to, all from the op-code. Note that the CLR instruction can also use other addressing modes to clear a memory location.

**Register** - This mode is used by only a handful of instructions. It is used whenever registers are the only operands but the opcode does not specify which register(S) to use (as in Inherent). In this mode the instruction byte is

followed by a 'post-byte' which is like an operand as it tells the CPU which registers to use. There are two groups of instructions which use Register addressing. They are EXG & TFR and PSH & PUL. The EXG and TFR instructions need two registers on which to operate, so each nybble of the post-byte describes a register. The PSH and PUL instructions can use any of the eight registers (excluding the stack pointer being used), so each bit of the post-byte represents a register (SET means use this register, RESET means don't use). An assembler knows that this group of instructions uses this mode and so no special symbol is needed to signify register addressing.

Immediate - This is the mode which is used like a constant in BASIC, that is the number to use is part of the program. When using Immediate mode addressing the instruction byte is IMMEDIATELY followed by the number that the instruction is to use. For example, LDA #10 is the Load A accumulator with the value 10. The "#" sign before the operand is used by most assemblers to denote Immediate mode.

Relative - This mode is used by the branch instructions in the 6809 set. Branch instructions are like GOTO's in BASIC but instead of giving the address (line number in BASIC) to branch to, an offset RELATIVE to the current value of the PC is given. This enables us to write Position Independent Code (how and why we write this type of code is covered later). The offsets used by the branch instructions can be either 8-bit or 16-bit signed numbers, 8-bit for small jumps (saves code i.e. 1 byte operand instead of 2) and 16-bit for long jumps.

The rest of the addressing modes use various methods of obtaining values out of memory, to be used with the instructions.

Extended - Extended addressing is the most straight forward addressing mode in this group. The 16-bit (two byte) operand following the instruction is an address which tells the CPU where the number it needs is located in memory. For example LDA \$2000 is load the A accumulator with the contents of memory location \$2000 (the "\$" sign just signifies a hex number). To continue the analogy with BASIC, this mode is like a variable i.e. you can assign and retrieve the values to or from a particular location (variable). Most assemblers do not need to be told that Extended addressing is to be used as it is used by default, although they can be forced into it.

Direct - This is a limited form of Extended addressing as it only requires a single byte operand to follow the instruction. This saves space and also running time of the program while having much the same effect. With Direct addressing one "page" (255 bytes) of memory can be accessed. This is also called 'zero page' addressing as in most micros this form of addressing can only access the 'zero page' (the

first 255 bytes of memory). The advantage of the 6809 over these other micros is that the 'zero page' can be moved. As was discussed before the DP (Direct Page) register contains the 'page number' that Direct addressing is currently using. In reality the CPU uses the DP register as the high byte of the Effective Address (the actual address the CPU will use) and the operand following the instruction as the low byte, e.g. LDA \$00 when the DP contains \$20, loads the A accumulator with the contents of \$2000. Good assemblers should be intelligent enough to choose between Direct and Extended modes but must also be able to be forced into one or the other. Whether or not you need symbols and what they are will depend upon your assembler.

**Indexed** - Now we're getting into the heavy stuff! Indexed addressing involves using a 16-bit register as a base pointer. A base pointer is just an address, except that in Indexed mode, we let the 6809 use this address as a reference point, and use various "indices" or offsets from that base to calculate the Effective Address.

The uses of Indexed modes are too numerous to detail in this book but as you become more familiar with machine language programming and your programs become bigger and more complex the beauty of these modes will become obvious.

As I said before there are numerous different types of Indexed modes and when Indexed addressing is specified in the op-code the CPU looks for the 'post-byte' which describes the type of Indexed addressing to use. The meaning of the post-byte can be seen in Appendix H (page 251).

a) Constant Offset from Register (Two's complement signed offset) - This mode involves adding a signed offset to a specified index register. There are four different ways of doing this.

i) No Offset - Just that! The value of the Index register we're using is the Effective Address. For example, LDA ,X loads the A accumulator from the contents of the address which is in the X index register.

ii) 5 Bit Offset - With this we can use offsets of -16 (\$10) to +15 (\$0F) i.e. 5 bits, including 0, from our index register to get the address we want. For example, LDA \$10,X loads the A accumulator with the contents of the address calculated by adding \$10 to the X index register. In this mode the offset is included in the post-byte.

iii) 8 Bit Offset - With this we have a much larger range of -128 (\$80) to 127 (\$7F) i.e. 8 bits. However, this form is slower, and takes up another byte of memory than the 5 bit mode.

iv) 16 Bit Offset - Now with this we have the ultimate range of -32,768 (\$8000) to 32,767 (\$7FFF) from our index register. This is even slower and longer than the 8-bit version as it needs 2 bytes just for the offset!.

b) Accumulator Offset from Register (Two's complement signed offset) - This works in a similar fashion to the 8 and 16 bit offset modes above. Only this time the signed offsets we are dealing with, are actually the values of certain registers. In fact these offset registers happen to be our accumulators A, B and D. The A and B registers are used for 8-bit offsets, and the D (A and B together) is used for 16-bit ones. This is extremely useful when accessing arrays as the offset into the array can be calculated in an accumulator and then only one instruction is needed to access it.

c) Auto Increment/Decrement Register - This is a very useful mode, which takes care of some of the drudgery involved in using pointers. After you've finished looking at one location, you quite often want to look at the next location following. To do this, you would normally have to add one to your pointer manually. However the 6809 can "auto-increment" your pointer, after you've looked at a location. For example, LDA ,X+ will load the A accumulator from the memory location in X then increment X so that it points to the next memory location. This saves us having to do it ourselves later on which saves us time when writing a program, it's also quicker for the 6809 to execute, and even takes up less room! The 6809 will also double increment (add two to the index register) for those occasions where you are handling 16-bit numbers.

Decrements can be done too. However, they work a little differently to increments. Auto-Incrementing increments the pointer AFTER the main instruction has been executed (e.g. after the operand has been loaded from memory), whereas Auto-Decrementing decrements the pointer BEFORE the main instruction has been executed, therefore we have post-increment and pre-decrement.

d) Constant Offset from Program Counter - This mode may not seem terribly useful at first, but it is, believe me. It along with Relative addressing (with branches) is necessary for Position Independent Code. The operand specifies an 8 or 16-bit offset which is added to the value of the PC as that instruction gets executed. This gives an address which, if you move your program (the PC is started at a different place) moves with it. For example, LDA \$20,PCR will load the A accumulator with the contents of the memory location whose address is calculated by adding the PC and \$20.

e) Indirect - This feature applies to nearly all the indexed modes mentioned above. Each of those modes eventually gives you a working address to play with. When indirection is used on these modes, the CPU takes the contents of the working address to be the address we want to use! Wheels within wheels! For example, LDA (,X) (the brackets indicate Indirect

mode) loads the A accumulator with the contents of a memory location whose address is in memory at the location pointed to by the X index register. Exceptions are the 'auto-increment/decrement by 1' modes (addresses use 2 bytes, so incrementing by 1 byte won't really do anything useful anyway!).

f) Extended Indirect - This mode is much the same as the Indirect mode above except that Extended addressing (instead of Indexed) is used to indicate where the Effective Address is in memory.

So there we have the 6809's addressing modes in all their glory! Pretty impressive, especially when you start comparing the 6809 with other 8-bit micros around. I suggest that you get familiar with all these modes as they are what makes the 6809 so nice to use. Probably the easiest way to get the hang of the various modes is to use the demonstration programs which load A using them (programs 2-4).

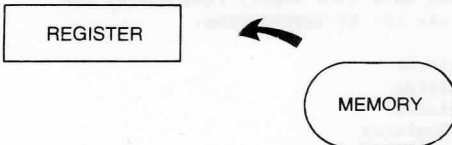
## CHAPTER 8

# Easy

Let's have a look at some of the simple instructions which almost every assembler program uses. In fact, there aren't too many programs that do not use them!

LD - Load Register

Say we want to put a value or the contents of a memory location into a register:



To do this we use the LD instruction. To load the A register we use LDA. For X it would be LDX, etc.. In fact, almost any of the 6809's registers can be loaded this way. The valid ones are:

LDA	A Accumulator
LDB	B Accumulator
LDD	D Accumulator
LDX	X Index Register
LDY	Y Index Register
LDU	U Stack Pointer
LDS	S Stack Pointer

These instructions let us load a value into a register. To determine what value it is we are loading, we need to remember the addressing mode we are using at the time.

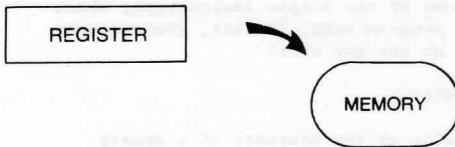
Also, we have 8-bit loads, and 16-bit loads. The 8-bit loads are for the 8-bit registers, and the 16-bit loads for the 16-bit ones, the 6809 knows how many bits to load by the register it is using.

You may be wondering how the 6809 loads a 16-bit value from what is basically one address. What actually happens is that the 6809 uses two addresses. From the address you give it, it gets the Most Significant Byte of the 16-bit number you are loading. Then it gets the Least Significant Byte from the next location in memory. Let's try a few of the more common ones:

LDA	#10	Put decimal 10 in accumulator A
LDX	\$1000	Load X from locations \$1000 and \$1001
LDX	,Y	Load X from locations Y and Y+1

See chapter 14, programs 2 - 4 for further examples of the LD instruction.

ST - Store Register



If we can load a value into a register, it would make sense to be able to store values back into memory from one of the registers. For this, we use the ST instruction.

STA	A Accumulator
STB	B Accumulator
STD	D Accumulator
STX	X Index Register
STY	Y Index Register
STU	U Stack Pointer
STS	S Stack Pointer

Just like the LD instruction above, we need to say what addressing mode we're using...

STA	\$1000	Put the contents of A in location \$1000
STX	,Y	Put the contents of X in locations Y and Y+1

See chapter 14 program 6 for a further example of the ST instruction.

## TFR - Transfer Registers



You may have noticed that there is no direct way of loading or storing the DP, CC and PC registers. Luckily we can indirectly load these registers, by using the TFR instruction. For example, to load the value \$10 into the DP register, we have:

LDA	#\$10	Load A with the desired contents of the DP register
TFR	A,DP	Transfer the contents of A to the DP register

Any register can be loaded indirectly using the TFR instruction, although we would normally only do this for the DP and CC registers. The PC register is a special case, and there are other ways of loading it

See chapter 14 program 5 for an example of the TFR instruction.

## ADD - Add to Register



Computers are meant to be able to add numbers, and this is the instruction that exercises the nitty-gritty of the 6809. There are three basic ADD instructions (along with all of their addressing modes of course).

ADDA	A Accumulator
ADDB	B Accumulator
ADDD	D Accumulator

The first two are 8-bit additions, and the ADDD is a 16-bit addition. To understand how they work, assume that you already have a number sitting in an accumulator. Then you use the ADD instruction, which gets the number you want to add from its operand (don't forget the addressing mode!) and adds it to the accumulator, leaving the result in the same accumulator. For example:

LDA	#\$10	Intialize A with hex 10
ADDA	,X	Add the value pointed to by X into A



And say that X pointed to location \$1000, and \$1000 contained \$56, then we would have \$56 + \$10 or \$66 in accumulator A.

See chapter 14 program 10 for further examples of the ADD instructions.

SUB - Subtract from Register



The 6809's SUB instruction works in much the same way as an ADD instruction, except in reverse. Instead of adding, we subtract. The three main subtract instructions work on the Accumulators:

SUBA	A Accumulator
SUBB	B Accumulator
SUBD	D Accumulator

For instance:

LDD	#1000	Load 1000 into Accumulator A
SUBD	,Y	Subtract 16-bit number pointed to by Y

Say that Y is pointing to \$2000, and at \$2000 we have \$01, and at \$2001 we have \$56. That's 1000 - \$0156.

See Chapter 14 program 16 for a further example of the SUB instruction

## CHAPTER 9

# Handy

Let's get down to actually writing some assembly programs. In our first attempt we'll write a 6809 program which will take two numbers, add them together, and store the result somewhere. Not that earth shattering, but then, Rome wasn't built in a day. Let's see what the piece of code we want would be:

LDA	NUM1	Get the first number
ADDA	NUM2	Add in the second number
STA	RESULT	Save the result somewhere

That might look like all we need, but it isn't. For one thing, we haven't said where NUM1, NUM2, and RESULT are in memory. For argument's sake, we'll put NUM1, NUM2, and RESULT immediately after the program:

LDA	NUM1	Get the first number
ADDA	NUM2	Add in the second number
STA	RESULT	Save the result somewhere

NUM1 RMB	1	place to get first number
NUM2 RMB	1	place to get second number
RESULT RMB	1	place to put answer

Note that RMB is not really a 6809 instruction. It is what's known as a "Pseudo-Op". It gets this name because it looks like an op-code, just like our normal 6809 mnemonics, but in fact isn't. Pseudo means false, so RMB is a false op-code! Pseudo-ops exist so that we can tell the assembler how we

want our program put together. The RMB (Reserve Memory Bytes) Pseudo-op just tells the assembler that we want to keep data (a number) at this location and the operand specifies how many bytes to reserve. Even if you don't have an assembler, it's good practise to use Pseudo-ops like this to show what you want.

In each case we reserve one byte for each number. This makes sense, because we are doing 8-bit or one byte addition. If we were doing 16-bit or two byte addition, we would use RMB 2 instead.

We have what you may think is the finished article. But it isn't! If you look a bit closer at our program above, you may see something wrong. Very simply where does it end? As it is, the 6809 would do the addition and keep going - including trying to execute our data areas we just so carefully defined.

It is at this point you would appreciate the hit/miss nature of assembly programming. What would the 6809 make of our data areas, thinking it is just another program? The answer is anything! Physically, your Dragon wont leap into the air and become the centre of an enormous explosion. Then again, because the 6809 has become a rogue, and is executing things it should not be, it will probably destroy any useful information you have in the machine. And losing information generally means losing work! So be aware that trepidation is called for.

Time to spill a bit more about how we will use this program. Because it is only a little one we will execute it from the Dragon's BASIC. This can be done via the EXEC or USR() statements in BASIC. The rules for using these statements are that your assembly language program should end with an RTS instruction. This may seem highly secretive to you at the moment, but we are not going to explain RTS's to you yet.

LDA	NUM1	Get the first number
ADDA	NUM2	Add in the second number
STA	RESULT	Save the result somewhere
RTS		
NUM1 RMB	1	place to get first number
NUM2 RMB	1	place to get second number
RESULT RMB	1	place to put answer

We still haven't said where we want our program to run. "In the computer!" some bright spark shouts. It is all very well putting the program in the Dragon, but we also need to give the addresses of the locations where the program is meant to be. To do this we use yet another pseudo-op ORG. ORG gives the origin of our program. This is the address of the first location of our program. In our example, we will use an origin of \$4000.

```

ORG      $4000      Starting address of program.

LDA      NUM1       Get the first number
ADDA     NUM2       Add in the second number
STA      RESULT     Save the result somewhere
RTS

```

```

NUM1 RMB    1       place to get first number
NUM2 RMB    1       place to get second number
RESULT RMB  1       place to put answer

```

But we still haven't got a 6809 machine code program that our Dragon can execute. This assembly language program could be turned into machine code, with very little pain by using an assembler program. However, programmers have masochistic natures, and this is what we are going to do. We are going to assemble it by hand. To do this let's rewrite the program so that it is on a piece of paper like this:

Address	Code	Label	Mnemonic	Operand
=====	=====	=====	=====	=====
			ORG	\$4000
			LDA	NUM1
			ADDA	NUM2
			STA	RESULT
			RTS	
		NUM1	RMB	1
		NUM2	RMB	1
		RESULT	RMB	1

The ADDRESS and the CODE columns are going to be filled in by the assembler, in this case you. Let's assume that we are going to use extended addressing in all of the instructions above, except for the RTS. If we look up the instruction descriptions in the back of this book, we find:

```

LDA
===
code   cyc   byt
Extended $B6/182  5   3

```

This means that an LDA instruction using extended addressing will use 3 bytes of memory, and will have a machine code equivalent of \$B6, or 182. Don't worry about the "cyc" or the "5" yet, they're not important right now. What is important are the op-code and the number of bytes used. Next to the LDA instruction in our program, we can put:

Address	Code	Label	Mnemonic	Operand
=====	=====	=====	=====	=====
\$4000	\$B6\$--\$--		LDA	NUM1

We also know that the next instruction (the ADDA) must begin at \$4003, hard on the heels of the LDA instruction. You will no doubt have noticed that we have not filled in the other two bytes that are part of the LDA instruction. This is because we are using extended addressing, and we must supply the address of the location we want to get that number from along with the \$B6 op-code for the LDA, but because, we haven't found out where NUM1 is supposed to be yet, we'll have to let those bytes lie, and come back to them later. Let's look forward to what we'll eventually have if we continue in this fashion:

Address =====	Code =====	Label =====	Mnemonic =====	Operand =====
\$4000	\$B6\$--\$--		LDA	NUM1
\$4003	\$BB\$--\$--		ADDA	NUM2
\$4006	\$B7\$--\$--		STA	RESULT
\$4009	\$39		RTS	
\$400A	\$XX	NUM1	RMB	1
\$400B	\$XX	NUM2	RMB	1
\$400C	\$XX	RESULT	RMB	1

Now we can see where NUM1, NUM2 and RESULT are in memory. Those \$XX are only there to show that NUM1, NUM2 and RESULT each take up one byte of memory, the contents of which we don't know. So now we go through and fill in those missing addresses from our instructions:

Address =====	Code =====	Label =====	Mnemonic =====	Operand =====
\$4000	\$B6\$40\$0A		LDA	NUM1
\$4003	\$BB\$40\$0B		ADDA	NUM2
\$4006	\$B7\$40\$0C		STA	RESULT
\$4009	\$39		RTS	
\$400A	\$XX	NUM1	RMB	1
\$400B	\$XX	NUM2	RMB	1
\$400C	\$XX	RESULT	RMB	1

The next step would be to enter the hex values in the code column into memory and execute the program. This can be done using a BASIC program. The start of this BASIC program is the "loader" found on page 128 and the rest is as below:

```

80 INPUT "ENTER A VALUE FOR 'NUM1'",A:POKE &H400A,A
90 INPUT "ENTER A VALUE FOR 'NUM2'",B:POKE &H400B,B
100 EXEC &H4000
110 PRINT "THE VALUE OF 'RESULT' IS NOW ";PEEK(&H400C)
120 GOTO 80
130 DATA B6, 40, 0A, BB, 40, 0B, B7, 40, 0C, 39, **

```

So there it is! Your first 6809 assembler program. Not really, We wrote it. Nevertheless, you will soon be able to write your very own 6809 programs.

## CHAPTER 10

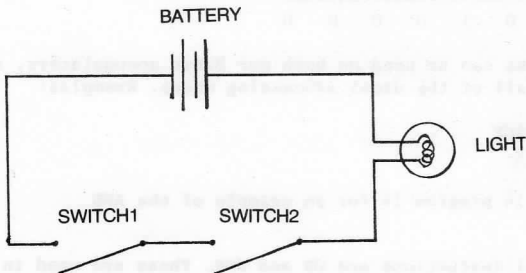
# Lets Get Logical

In this chapter we will examine the logical operations which the 6809 is capable of performing. Specifically they are:

- AND - AND
- EOR - Exclusive OR
- OR - OR

These operations are really quite simple, but you need to get wise to them if you want to use your 6809 to its full potential.

If you are not familiar with ANDs, OR,s etc., take a look at the electric circuit below:



Diag.5

It's pretty obvious how this circuit works. Unless we have both Switch 1 and Switch 2 in the ON position, the light bulb won't go. To describe the operation of this circuit we can use what is called a truth table. This is just a table which says what the light bulb does for various combinations of switch 1 and switch 2:

Switch 1	OFF	ON	OFF	ON
Switch 2	OFF	OFF	ON	ON
-----				
Result	OFF	OFF	OFF	ON

This is an AND operation at work here. Both switch 1 AND switch 2 must be ON for the light to be ON also. Now if we were dealing in binary digits, then we would get a result of 1 only if you AND 1 with 1. i.e.

Operand 1	0	1	0	1
Operand 2	0	0	1	1
-----				
Result	0	0	0	1

Say we have a hex number \$5B in our accumulator A. In binary this would look like:

0 1 0 1 1 0 1 1

Say also that we want to AND this number with another hex number, this time \$0D:

0 0 0 0 1 1 0 1

To find out the result we AND the two numbers one bit at a time. First we AND the two bit-7's, then the two bit-6's and so on, until all 8 bits have been ANDed together. So we get:

0 0 0 0 1 0 0 1

Here is another example:

	0	1	0	1	1	0	0	0
AND	1	1	0	1	0	1	1	0
-----								
=	0	1	0	1	0	0	0	0

AND operations can be used on both our 8-bit accumulators, A and B, with all of the usual addressing modes. Examples:

ANDA #\$OF  
ANDB ,X+

See Chapter 14 program 18 for an example of the AND instruction.

Other logical instructions are OR and EOR. These are used in the same fashion as the AND instruction. OR is simple but EOR is a little harder to understand.

EOR, Exclusive OR, has the function that if a bit is set in either operand only then is the result bit set, but if neither or both of the bits are set then the result is reset.

In other words of the two bytes all bits in the same position in the byte yield a 0 if they are the same and a 1 if they have different values.

Here are some examples for EOR:

```

EOR   1  1  0  1  1  0  0  0
      1  0  1  1  0  1  0  0
-----
=     0  1  1  0  1  1  0  0

```

```

EOR   1  0  0  1  1  0  1  1
      0  1  0  1  0  0  1  0
-----
=     1  1  0  0  1  0  0  1

```

OR sets the result bit if a bit is set in operand 1 OR in operand 2. Using OR the only time the result bit is reset (0) is when the bit in both operands is reset.

To put it simply, if neither bit is set the result is 0 otherwise it is 1.

Examples of OR:

```

OR    1  1  0  1  1  0  0  0
      1  0  1  1  0  1  0  0
-----
=     1  1  1  1  1  1  0  0

```

```

OR    1  0  0  1  1  0  1  1
      0  1  0  1  0  0  1  0
-----
=     1  1  0  1  1  0  1  1

```



## CHAPTER 11

# Condition Codes

If you have done much programming in BASIC, you will know that a program which doesn't make decisions is a very simple program indeed. The add-two-numbers program we looked at in the previous chapter is, by this definition, a very simple program. In BASIC we can make decisions by using the IF statement. In the 6809 assembly language we have no such thing as an IF statement. However, we still have ways of making decisions on the 6809.

In order for you to understand how decisions are made by the 6809 you really need to know what the flags inside the Condition Code register (CC) represent. So let's take a look at what the CC register contains.

7	6	5	4	3	2	1	0
E	F	H	I	N	Z	V	C

In the chapter on the 6809, I said that the CC register was a collection of flags and switches living in an 8-bit register. Flags are "indicators" showing that something is either ON or OFF. So obviously this type of status can be represented by a single bit. These flags show that particular conditions have arisen in the 6809, and provide us with some feedback from operations we have performed. So let's take a look at what the status bits mean.

C - Carry Flag. The concept of the carry flag may seem a little alien to you at first, but in a little while you will realise that you have already been using a carry flag of your own for years! Think of when you add two numbers together by hand.

```
  4
+ 8
--
==
```

The answer is 12 of course, but how do we get 12 for our answer? What actually happens is that we add 4 and 8 together, and get the following:

```
  4
+ 8
--
  2
==
```

But we also have to carry 1 from the ones column into the tens column. That's how we get:

```
  4
+ 8
--
 12
==
```

So now we look at the fairly typical situation where we have to add two unsigned hex numbers on the 6809.

```
LDA #$17
ADDA #$EA
```

We would find that A would contain \$01 after the ADDA instruction. Obviously \$17 + \$EA should be more than this, in fact we would really expect it to be \$101. That takes more than 8 bits, and therefore our result of \$101 is cut back to the 8-bit number \$01. However, that extra \$100 has not been entirely lost. If we looked at the carry flag after the ADDA instruction, we would find that it is SET. This shows that our result was too big to fit in an 8-bit unsigned number. Remember I am talking about unsigned numbers here. This carry bit can now be used in subsequent calculations. i.e. to do a 16-bit addition:

```
LDA #$17
ADDA #$EA
STA RESULT + 1
LDA #$0
ADCA #$0
STA RESULT
```

The ADC is really just an add instruction, that also adds in the carry which may or may not have been set from a previous add instruction. This use of carry lets us add (or subtract) very big numbers (8 bits, 16 bits, 32 bits, etc.). If carry is SET after a subtraction then we had to "borrow" 1 from the next byte.

The carry flag is not only used for arithmetic though. It is also used in bit manipulation instructions called shifts. These instructions are quite simple and are used for multipling and dividng by two. Let's see how simple, and put the 8-bit number \$37 in A:

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

Now consider the effect of an LSRA (Logical Shift Right on A) instruction. All this does is move everything one bit position to the right:

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

So a bit falls off the right-hand (least significant) end, and a gap is left at the left-hand (most significant) end. As it happens, the 6809 will use a 0 to fill that gap and the number has been divided by two:

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

But what happened to the 1 that got pushed off the end? It fell into the carry flag in the CC register. There are several of these bit manipulation instructions and most involve a carry in ways like this.

V - Overflow Flag. Overflow works a bit like carry does, only this time its for signed numbers. Put it this way. Say we have the signed 8-bit number 127 or \$7F which we all know is the largest positive number we can have using 8 bits.

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

Now let's add 1 to this number.

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

It seems that we now find ourselves with -128, using SIGNED numbers. Clearly we have been lumbered with an incorrect result. However, as a result of this operation, the V flag will be set, indicating that our result "overflowed" into the sign bit of our number.

Z - Zero Flag. This is an easy one to understand. Anytime you perform an operation which gives a result of 0, the Z flag will be set. For example:

```
CLRA  
LDA #0
```

would both set the Z flag. Any operation which leaves a result of 0 in a memory location, or a register will set Z - regardless of any carry, half-carry or overflow.

N - Negative Flag. This is also an easy one to understand. If you perform an instruction which leaves the MSB (or sign bit) SET, the N flag will also be set. This is to indicate that the result is negative, if you are using signed numbers. It does not matter if there is also a carry, half-carry or an overflow set. Clearly Z and N cannot both be set at the same time, as a result cannot be both zero and negative at the same time.

I - IRQ Mask. This is one the switches that live in the CC register. It is not really relevant at this stage of the book and I won't say any more about it until a later chapter.

H - Half-carry Flag. If you add two numbers together, and you get a "carry" from bit-3 to bit-4 in your addition, then the H flag will be set. For instance, if you have \$0F in A and then you add 1 to A:

```
LDA #$0F
 7 6 5 4 3 2 1 0
 0 0 0 0 1 1 1 1
```

This will give us a result of \$10 in accumulator A:

```
ADDA #$01
 7 6 5 4 3 2 1 0
 0 0 0 1 0 0 0 0
```

It will also cause a carry to be brought from bit-3 over to bit-4 of A. This will cause the half-carry flag to be set. The H flag does not have too many uses, and is in fact only used for performing BCD (Binary Coded Decimal) arithmetic.

F - FIRQ Mark. Once more you will have to wait until the chapter on interrupts to learn about this one.

E - ENTIRE Flag. Once again you will have to wait until the interrupts chapter.

## Decisions, Decisions

You know all you need to know about the flags in the CC register. But how does this give you the ability to write a program which will make its own decisions?

The 6809 has several instructions called "conditional branches" which allow us to check for various combinations of these flags, and if applicable then the 6809 will branch to another part of our program. This works in much the same as the IF-THEN-GOTO statement in the Dragon's BASIC. Here is a list of the 6809's conditional branches:

BCC  
BCS  
BEQ  
BGE  
BGT  
BHI  
BHS  
BLE  
BLO  
BLS  
BLT  
BMI  
BNE  
BPL  
BVC  
BVS

For instance, if we want to branch to a new part of our program called ZPART, if the Z flag is set, then we would write:

BEQ ZPART

How do we set up the condition codes in the first place, after all, the Z flag isn't about to set itself to please our program. There are some extra instructions which we can use in testing:

CMP  
BIT  
TST

CMP - This works very much like a SUB instruction, except that you can use CMP on any register except DP, CC or PC. The CMP instruction causes the 6809 to subtract the operand from the register being compared. This will set the CC register in exactly the same fashion as a normal SUB would, however, unlike the SUB instruction, the result of the subtraction is not saved back into the register. Instead it is discarded, thus leaving the register uncorrupted. If we were comparing a register against several different values, this would save us reloading the register after each CMP instruction.

See Chapter 14 Program 22 for an example of the CMP instruction.

## Loops

Looping is one of the most often used techniques you will ever see in programming, no matter what language you decide to use. Looping in 6809 assembly language is pretty nifty. The main part of any loop is a value called the loop count. This is the value that tells us how many times we have gone through a loop. By convention, we use the B register as the loop counter. In almost every case you could use the A

register with no problems and if we ever get into a situation where the B register has to be used for something else, we can use A.

In BASIC we have the FOR-NEXT loop statement. Here are some 6809 programs which will do much the same thing as FOR-NEXT loops:

```
FOR I = 1 TO 100:  
NEXT I
```

```
      LDB #1      load initial value of loop counter  
LABEL1 NOP      repeat some action  
      .  
      .  
      INCB      next loop-count value  
      CMPB #100  have we reached end of our loop?  
      BHS LABEL1 no, then go back again.
```

```
FOR I = 100 TO 1 STEP -1:  
NEXT I
```

```
      LDB #100   load initial value of loop counter  
LABEL1 NOP      repeat some action  
      .  
      .  
      DECB      decrement our loop counter  
      CMPB #1    have we reached end of our loop?  
      BNE LABEL1 no, then go back again
```

Note the signed test done in the last example. Actually, these structures are not used all that often in assembly language, even though they are handy to know. For example:

```
      LDB #1      load the initial value  
LABEL1 NOP      repeat some action  
      .  
      .  
      INCB      next loop count value  
      CMPB #7    have we finished the loop?  
      BLS LABEL1 no, then loop back until done
```

If we have to go through the loop 7 times and it doesn't matter which way the loop variable goes (1 to 7 or 7 to 1), we can use instead:

```
      LDB #7      initialise the loop counter  
LABEL1 NOP      repeat some action  
      .  
      .  
      DECB      count down instead of up  
      BNE LABEL1 Z will be set if finished
```

This version is shorter and quicker because we don't need the CMP instruction. When DECB causes B to hit 0, the Z flag will automatically be set and the loop will end.

Often we also want to use the B register, i.e. the loop counter, as an offset for use in a register offset indexed mode instruction (like LDA B,X). Say we had our X register pointing to the start of a data list:

```
X   Item 1
X+1 Item 2
X+2 Item 3
X+3 Item 4
X+4 Item 5
```

Also assume that the order of access doesn't matter. To access these data items we would use:

```
      LDB #4      count down from X+4
LABEL1 LDA B,X   get next data item
      .
      .
      .
      DECB      move to next data item in list
      BPL LABEL1 finish after X+0 is done
```

This will access all of the locations in that data list.

These examples all use loop counters that are only 8 bits long (accumulators A and B). If we need to use bigger numbers as loop counters, obviously we can't use the A or B registers because they are only 8-bits long. We therefore use our 16-bit registers: D, X, Y or U. For instance:

```
FOR I = 500 TO 999:
NEXT I

      LDX #500   load X with initial value
LABEL1 NOP      repeated part of loop
      .
      .
      .
      LEAX 1,X   increment X register
      CMPX #1000 have we finished yet?
      BNE LABEL1 no, then loop back
```

The new instruction LEA (Load Effective Address) is a neat way of letting us adjust and even calculate pointers easily. The "LEA 1,X" instruction means: "get the value currently in the X register, adjust it by +1, and store the result back into register X". This is an exceptionally powerful instruction to have, as it effectively lets us do 16-bit arithmetic on registers other than the accumulators. It's not quite the same. An LEA instruction will not set up the CC register flags in the same way that arithmetic on the

accumulators will. In fact the only flag LEA does effect is the Z flag, and even then that is only for LEAX and LEAY. The LEA instruction has only one mode - indexed. We could have used the D register as our loop counter, but instead of the LEA we would have to use:

```
ADDD #1
```

as there is no LEA instruction available on D register. Another example might be:

```
FOR I = 0 TO 1000 STEP 5:
NEXT I

        LDX #0      load the initial loop count
LABEL1  NOP        repeat some action
        .
        .
        .
        LEAX 5,X    adjust X by step value
        CMPX #1005  have we finished looping yet?
        BLS LABEL1 no, then loop back?
```

Say we need to use the D register as a 16-bit offset, in much the same way that we used B as an 8-bit offset, in the example using the list. However, the list is much bigger, with 500 2-byte items:

```
X   Item 1
X+1
X+2 Item 2
X+3
X+4 Item 4
X+5
etc.
```

We could use something like the following:

```
        LDD #0      initialise D to start of list
LABEL1  LDY D,X     do something with the data
        .
        .
        .
        ADDD #2     point to next data item
        CMPD #500   have we finished looping yet?
        BNE LABEL1 no, then loop back?
```

Now even though this is quite valid, you will find it is not a good programming technique. This is because it ties up both accumulators - and accumulators are the most powerful registers and they should not be wasted. It would actually be better to use:



```
LDD ,X++    get a data item from the list
.
.
.
CMPX #ENDLIST have we reached the end of the list
BNE LABEL1  no, then keep looping
```

So with the 6809 we can have both short (8-bit) or long(16-bit) loops.

## CHAPTER 12

# Stacks and Subroutines

Everyone has come in contact with stacks, nobody has escaped using them. All the way from the high-powered business executive down to the dish-washer. The business man makes neat piles of paper on his desk, while the dish-washer dries his plates and piles them on the table. Stacks, piles, heaps, whatever. They are all the same thing. They are intermediate stopping places used for temporary storage, whether we are talking about paper, plates, or data.

The 6809 uses stacks, as do most micro-processors. The reason why they need stacks may not seem obvious at first, so just imagine the following.

Say we have a business executive working on a job, A, which is not all that important. His boss rings up with a new job, B, which has to be finished by the end of next week. Our executive, being a tidy worker, puts all the information relating to his current job in a pile to one side of his desk and gets out the file for the new job. As he works his way through job B the boss rings up with another new job, C, which has to be completed by tomorrow. So our businessman, piles up all the information for job B and puts it on top of job A. He then gets out the file for job C and starts to work on that.

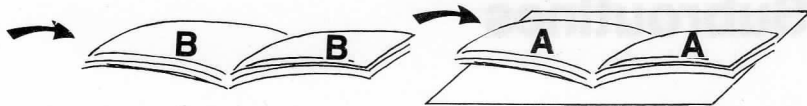
As he finishes each job, and puts the file away, he will automatically pick up the file currently sitting on the top of his stack, and resume working on it. In this case, when he

is done with job C's file, he will pick up B's. After B's file, he will come back to the file he was originally working on. Thus he can maintain his reputation as a neat and tidy executive.

Now let us take a look at how his stack is changed by all this. First of all he is using his file, or data, on job A. He has no previous files piled up in his stack:



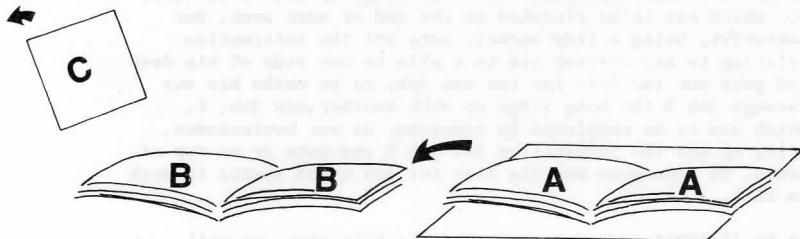
Now he decides to use file B, and in the meantime, "pushes", or saves file A on his stack:



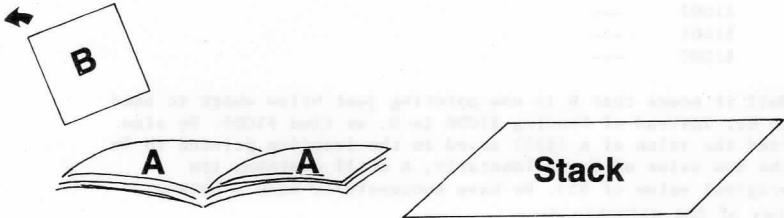
And now he saves file B on the stack, and gets out file C:



Now he starts "pulling" files back off the stack. He is finished with file C, so he puts it away again. He then gets file B off the top of his stack:



Having done with file B, he sticks it back in his filing cabinet, and gets file A back off the stack, and resumes work on the original job. The stack is now empty.



By organising his work in a stack while he does other smaller jobs, the executive keeps things systematic, and thus preserves his reputation as a neat and tidy worker.

In much the same manner, the 6809 preserves its reputation as an orderly CPU by maintaining a stack which is controlled by the S register. This is the time to reveal the PSH (Push onto stack) and PUL (PULL from stack) instructions! Consider the area of memory shown below:

```

$1007 ---
$1006 ---
$1005 ---
$1004 ---
$1003 ---
$1002 ---
$1001 ---
$1000 ---

```

Consider also that our S register, or Stack Pointer, contains the value \$1006. Now pretend that we have some valuable data in Accumulator A - \$55. Imagine that this value must not be lost or else the program will lose it's train of thought. Also, all of the other registers are tied up with necessary data. Things are pretty tightly stretched. As soon as we need to use a register for some new purpose, we are in trouble.

We now need to re-use Accumulator A. What do we do??? Simple. We "PuSH" A onto the stack like so:

```

PSHS   A

```

Let's have a look at the effect this has on our stack pointer and that area of memory:

\$1007	---	
\$1006	---	
\$1005	\$55	--- Stack Pointer now points here
\$1004	---	
\$1003	---	
\$1002	---	
\$1001	---	
\$1000	---	

Well it seems that S is now pointing just below where it used to be. Instead of finding \$1006 in S, we find \$1005. We also find the value of A (\$55) saved in the location pointed to by the new value of S. Incidentally, A still contains its original value of \$55. We have successfully made a backup copy of the valuable data.

A little further on in time, after Accumulator A has undergone some changes, we decide that it is time to get \$55 back into A. S will still be pointing to location \$1005. To get \$55 back into A again, we just "PUL" A off the stack, like so:

PULS        A

Our stack should now look like the following:

\$1007	---	
\$1006	---	
\$1005	\$55	--- Stack Pointer now points here
\$1004	---	
\$1003	---	
\$1002	---	
\$1001	---	
\$1000	---	

If we wanted, we could have Pushed or Pulled several registers at once. For instance:

PSHS CC,A,B,DP,X,Y,U,PC

would push all of the 6809's registers onto the stack, except for S. This makes sense, because there is little use in saving the location of the stack on the stack.

So far we have only looked at the S, or hardware stack. But there is another stack pointer in the 6809, the U or user stack pointer. The U stack can use the PSH and PUL instructions just the same as the S stack. Note that S can be saved on the U stack and U can be saved on the S stack but neither can be saved on their own stack.

The PSH and PUL instructions have an order of precedence. That is to say, The 6809 will save the registers you want to in a particular order. This order is fixed, so no matter what order you specify the registers, they will be PuShed in the following order:

Pushed onto stack first  
(highest address)

PC	low byte
PC	high byte
U/S	low byte
U/S	high byte
Y	low byte
Y	high byte
X	low byte
X	high byte
DP	
B	
A	
CC	

(Lowest Address)  
Pushed onto stack last

We can now use stacks to save registers, while we use the registers for something else. Now for a very fundamental use of stacks, which you will use time and time again ...

Let's say you have written a program which has to perform a complicated procedure twice, using different data. The natural approach would be to write the program so that it contains the code for that procedure twice, once for each occasion it is used. Obviously this will produce a lot of redundant code. In BASIC, you would group such pieces of common code together into subroutines, which you would then call with a GOSUB. This method saves a lot of room in BASIC.

The same goes for Assembly Language too. The 6809 is quite capable of handling subroutines thanks to the S stack. Consider the actions of calling a subroutine in BASIC. Say at line 100, we call a subroutine at 2000:

```
100 GOSUB 2000: REM CALL SUBROUTINE AT 2000
110 PRINT "FRIEND!"
120 END
2000 REM *** SUBROUTINE
2010 PRINT "HELLO ";
2020 RETURN
```

Our program will execute line 100, find the GOSUB and then jump to line 2000, and start executing from there. When it hits the RETURN at line 2020, the program knows it has reached the end of the subroutine, and resumes execution at the line AFTER the GOSUB in line 100.

But it has to save this line number somewhere, before it starts the subroutine. Because it is possible to call subroutines from within subroutines, stacks are ideal for the job.

Here is an Assembly Language example of a subroutine. Say we have a subroutine which will add 2 to Accumulator A.

```
$1000   LDA    #5      Put initial value into A
$1002   JSR    ADD2    Call subroutine to add 2
                        to A.
$1004   NOP
        .
        .
        .
$1010  ADD2  ADDA    #2      Add 2 to A.
$1012   RTS                      Return.
```

Even though this example is pretty trivial, it demonstrates the mechanics of a subroutine quite well. What happens when the 6809 executes this program is:

- a) Accumulator A is initialised with, in this case, 5.
- b) The ADD2 subroutine is called by the JSR instruction. This instruction does two things. Firstly, it gets the address of the instruction following the JSR, in this case, the NOP at \$1004. The 6809 then pushes this address onto the S stack. Secondly, the address of the ADDs subroutine is loaded into the 6809's PC (Program Counter) register so that the 6809 will then start executing the subroutine.
- c) The ADDA instruction adds 2 to Accumulator A.
- d) The 6809 now comes across an RTS instruction. An RTS causes the return address, pushed onto the S Stack by the JSR, to be pulled off and placed into the PC register. This causes the 6809 to resume execution of the program at \$1004.

So you have seen what a subroutine is. However, there is still more to learn, before you can use them properly. Here are some rules of thumb:

- 1) Parameters - A parameter is an item of data you pass to or from a subroutine for it to do work on. In our previous example, the subroutine uses Accumulator A as a parameter. It passed 5 as a parameter, and then it returned 7 or A+2.
- 2) Saving Registers - It is a good idea to make sure that any registers, which are not used to pass back parameters, are left unchanged. This eliminates a major source of errors in writing programs which use subroutines.
- 3) Workspace - You learnt before that PSH and PUL can be used to gain temporary workspace on the S stack. This is fine if you only do one or maybe two PSH's on your stack throughout your subroutine. However, if you have any more, things start to get complicated, and if you forget to PUL all the registers that you PSHed, the program will crash, as the 6809 will lose the address it needs to return from the subroutine.

The best way of getting a lot of temporary storage on the stack is to reserve some at the beginning of your subroutine by using the LEAS instruction. Thus:

```
LEAS      - #bytes, S
```

This causes the stack pointer to drop by the number of bytes specified. These locations are now free for you to use via the indexed addressing mode, e.g. STA 1,S.

Say we did a LEAS -5,S at the start of our subroutine. Our stack would look like this:

```
Workspace
Return Address
4,S
3,S
2,S
1,S
0,S - S points here
```

\*\*\* convert to diagram here \*\*\*

To unload the 5th storage location into A we would use:

```
LDA      4,S
```

Just before our subroutine finished, we would have to release the workspace with a

```
LEAS     5,S
```

Whatever you do, don't use locations outside those reserved as temporary storage! If you try to go higher you will very probably corrupt a return address sitting on the stack, and if you try to go below, i.e. -n,S, your storage will be overwritten when you call another subroutine.

4) Documentation - This does not sound important, but it is! Whenever you write a subroutine, you should include comments saying:

- a) Label / Description
- b) Function
- c) Parameters Passed
- d) Parameters Returned

Now let's bring these four elements together, and write ourselves another subroutine. In this subroutine, we will multiply Accumulator A by \$57 and then put the result in X. Apart from X, no other register should be changed. Here is the result.



```

*****
*
*   MUL57 - Multiply A by $57 and put result in X
*
*   Passed: A - Multiplicand
*
*   Returned: X - A * $57
*****

```

```

MUL57  PSHS   A,B,   Save registers
        LEAS  -2,S   Reserve workspace
        LDB   #57    Load multiplier into B
        MUL   ,S     A * $57
        STA   ,S     Save result into workspace
        STB   1,S    Put result in X
        LDX   ,S     Put result in X
        LEAS  2,S    Release workspace
        PULS  A,B    Re-store registers
        RTS

```

We didn't say this was the most efficient way to let  $X = A * 57$ ! It was just an example. If the MUL instruction is a mystery to you, relax. All MUL does is multiply the 2 8-bit Accumulators together (both as unsigned numbers) and leaves the 16-bit result in D.

And now for a little talk about the U register. You might infer from the examples in the text, that We don't take the U stack very seriously. The U stack CAN be used for temporary storage just like the S stack. But the S stack can do anything the U stack can, with proper management. So keeping this in mind, We prefer to use U as another Index Register. We suggest that you do the same.

Basically, that's it. There isn't really a lot to stacks, except being very, very careful.

## CHAPTER 13

# The 6809 Instruction Set

This section describes each of the 6809 instructions in detail. Each page contains the instructions in the format given below.

**INSTRUCTION MNEMONIC** - Brief description of the instruction.

**Function** - a formal definition of the instruction's function.

**Description** - a detailed description of the instructions function.

**Addressing Modes** - this gives the machine code for the instruction in both hexadecimal and decimal under the heading "code", the number of clock cycles needed to execute the instruction under the heading "cyc", and the number of bytes needed to hold the instruction and operand, in a program, under the heading "byt", for each of the addressing modes valid for that instruction.

**Condition Codes** - this defines how each of the bits in the Condition Code Register is affected.

Key to Characters Used

=====

- A Accumulator A
- B Accumulator B
- D Double Accumulator D (A & B)
- X Index Register X
- Y Index Register Y
- U Index Register/User Stack Pointer U
- S Hardware Stack Pointer
- PC Program Counter
- DP Direct Page Register
- CC Condition Code Register
- M Specified Address (Memory)
- ? Undefined
- \* No Change
- ! Change
- ^ Logical AND
- \_ Add
- + Add
- Subtract

CHAPTER 11

The 6809  
Instruction Set

ABX - Add Accumulator B to Index Register X Unsigned

Function: ABX  $X \leftarrow X + B$  (unsigned)  
" ←" This is an accepted convention,  
in BASIC you would use  $X = X + B$

Description:  
Add the contents of Accumulator B to those of Index Register X, storing the result back into the X Register. The number in B is treated as an unsigned 8-bit number.

Addressing Modes:

ABX  
===  
code cyc byt  
Inherent \$3A/58 3 1

Condition Codes:

	E	F	H	I	N	Z	V	C
ABX	*	*	*	*	*	*	*	*

ADC - Add with Carry

Function: ADCA A ← A + M + C  
ADCB B ← B + M + C

Description:

Add the contents of memory and the Carry Flag to either the A or B Accumulator. The result is stored in the stated accumulator.

Addressing Modes:

	ADCA		ADCB	
	====		====	
	code	cyc byt	code	cyc byt
Immediate	\$89/137	2 2	\$C9/201	2 2
Direct	\$99/153	4 2	\$D9/217	4 2
Extended	\$B9/185	5 3	\$F9/249	5 3
Indexed	\$A9/169	4+ 2+	\$E9/233	4+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
ADCA	*	*	!	*	!	!	!	!
ADCB	*	*	!	*	!	!	!	!

ADD - Add

Function:    ADDA    A ← A + M  
               ADDB    B ← B + M  
               ADDD    D ← D + M:M+1

Description:

Add the contents of the operand to either Accumulator A, B, or D. The result is stored in the stated accumulator.

Addressing Modes:

	ADDA =====			ADDB =====			ADDD =====		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Immediate	\$8B/139	2	2	\$CB/203	2	2	\$C3/195	4	3
Direct	\$9B/155	4	2	\$DB/219	4	2	\$D3/211	6	2
Extended	\$BB/187	5	3	\$FB/251	5	3	\$F3/243	7	3
Indexed	\$AB/171	4+	2+	\$EB/235	4+	2+	\$E3/227	6+	2+

Condition Codes:

	E	F	H	I	N	Z	V	C
ADDA	*	*	!	*	!	!	!	!
ADDB	*	*	!	*	!	!	!	!
ADDD	*	*	*	*	!	!	!	!

## AND - Logical AND

Function:    ANDA   A   ← A ^ M  
               ANDB   B   ← B ^ M  
               ANDCC  CC ← CC ^ IMM

### Description:

Logically AND the contents of memory with the specified Register. The result is stored in the stated register. The ANDCC (AND Condition Code register) instruction can use Immediate addressing only, and can be used to reset an interrupt mask or status.

### Addressing modes:

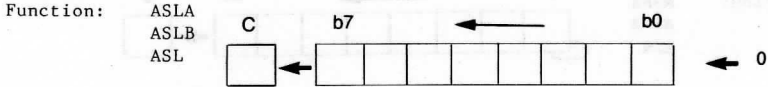
	ANDA ====			ANDB ====			ANDCC =====		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Immediate	\$84/132	2	2	\$C4/196	2	2	\$1C/28	3	2
Direct	\$94/148	4	2	\$D4/212	4	2			
Extended	\$B4/180	5	3	\$F4/244	5	3			
Indexed	\$A4/164	4+	2+	\$E4/228	4+	2+			

### Condition Codes:

	E	F	H	I	N	Z	V	C
ANDA	*	*	*	*	!	!	0	*
ANDB	*	*	*	*	!	!	0	*
ANDCC	?	?	?	?	?	?	?	?

(Changed according to operand)

ASL - Arithmetic Shift Left



Description:  
Shift the contents of memory, or either of the accumulators to the left one bit position, with bit 7 going into the Carry flag, and a 0 going into bit 0. The result is stored back in the specified memory location, or accumulator.

Addressing Modes:

	ASLA ====			ASLB ====			ASL ===		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$48/72	2	1	\$58/88	2	1			
Direct							\$08/8	6	2
Extended							\$78/120	7	3
Indexed							\$68/104	6+	2+

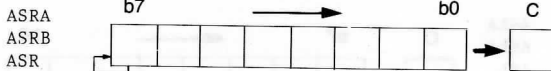
Condition Codes:

	E	F	H	I	N	Z	V	C
ASLA	*	*	?	*	!	!	!	!
ASLB	*	*	?	*	!	!	!	!
ASL	*	*	?	*	!	!	!	!



ASR - Arithmetic Shift Right

Function:



Description:

Shift the contents of memory, or either of the accumulators to the right one bit position, with bit 0 going into the Carry flag, and bit 7 staying as it was. The result is stored back into the specified memory location, or accumulator.

Addressing modes:

	ASRA ====		ASRB ====		ASR ===				
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$47/71	2	1	\$57/87	2	1			
Direct							\$07/7	6	2
Extended							\$77/119	7	3
Indexed							\$67/103	6+	2+

Condition Codes:

	E	F	H	I	N	Z	V	C
ASRA	*	*	?	*	!	!	*	!
ASRB	*	*	?	*	!	!	*	!
ASR	*	*	?	*	!	!	*	!

BCC - Branch on Carry Clear

Function: BCC goto specified address if C=0  
 LBCC

Description:

Test the Carry flag in the CC register. If C=1 then do nothing, and continue execution with the next instruction. If C=0, then branch to the current value of the PC (Program Counter) plus the signed displacement (-128 to +127 for BCC or -32767 to +32767 for LBCC). The L at the start of all branch instructions denotes a Long Branch (16 bit offset), and LBCC takes 6 cycles if the branch is taken and 5 if it isn't.

Note that whilst the instruction is being executed the PC is already pointing to the next instruction and this must be taken into account when calculating the offset.

	BCC		LBCC
	===		====
	code	cyc	byt
Relative	\$24/36	3	2
			\$10/16 5/6 4
			\$24/36

Condition Codes:

	E	F	H	I	Z	V	C
BCC	*	*	*	*	*	*	*
LBCC	*	*	*	*	*	*	*

## BCS - Branch on Carry Set

Function:    BCS    goto specified address if C=1  
              LBCS

### Description:

Test the Carry flag in the CC register. If C=0 then do nothing, and continue execution with the next instruction. If C=1, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BCS or -32768 to +32767 for LBCS). Note that the PC points to the instruction following the BCS whilst it is being executed and that LBCS takes 6 cycles if the branch is taken and 5 if it isn't.

### Addressing Modes:

	BCS			LBCS			
	===			=====			
	code	cyc	byt	code	cyc	byt	
Relative	\$25/37	3	2	\$10/16	5/6	4	
				\$25/37			

### Condition Codes:

	E	F	H	I	Z	V	C
BCS	*	*	*	*	*	*	*
LBCS	*	*	*	*	*	*	*

## BEQ - Branch if Equal to Zero

Function:     BEQ   goto specified address if Z=0  
              LBEQ

### Description:

Test the Zero flag in the CC register. If Z=0 then do nothing, and continue execution with the next instruction. If Z=1, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BEQ or -32768 to +32767 for LBEQ). Note that the PC points to the instruction following the BEQ whilst it is being executed and that LBEQ takes 6 cycles if the branch is taken and 5 if it isn't.

### Addressing Modes:

	BEQ				LBEQ				
	===				====				
	code	cyc	byt		code	cyc	byt		
Relative	\$27/39	3	2		\$10/16	5/6	4		
					\$27/39				

### Condition Codes:

	E	F	H	I	Z	V	C
BEQ	*	*	*	*	*	*	*
LBEQ	*	*	*	*	*	*	*

BGE - Branch on Greater than or Equal to Zero

Function: BGE goto specified address if N(XOR)V = 0  
LBGE

Description:

Test the Negative and Overflow flags in the CC register. If N(XOR)V=1, then do nothing and continue execution with the next instruction. If N(XOR)V=0, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BGE or -32768 to +32767 for LBGE). Note that the PC points to the instruction following the BGE whilst it is being executed and that LBGE takes 6 cycles if the branch is taken and 5 if it isn't.

Addressing Modes:

	BGE			LBGE			
	===			=====			
	code	cyc	byt	code	cyc	byt	
Relative	\$2C/36	3	2	\$10/16	5/6	4	
				\$2C/36			

Condition Codes:

	E	F	H	I	Z	V	C
BGE	*	*	*	*	*	*	*
LBGE	*	*	*	*	*	*	*

BGT - Branch on Greater than Zero

Function: BGT goto specified address if Z + N(XOR)V=0  
 LBGT

Description:

Test the Zero, Negative and Overflow flags in the CC register. If Z+N(XOR)V=1, then do nothing, and continue execution with the next instruction. If Z+N(XOR)V=0, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BGT or -32768 to +32767 for LBGT). Note that the PC points to the instruction following the BGT whilst it is being executed and that LBGT takes 6 cycles if the branch is taken and 5 if it isn't.

Addressing Modes:

	BGT				LBGT
	===				====
	code	cyc	byt		code
					cyc
					byt
Relative	\$2E/46	3	2		\$10/16
					5/6
					4
					\$2E/46

Condition Codes:

	E	F	H	I	Z	V	C
BGT	*	*	*	*	*	*	*
LBGT	*	*	*	*	*	*	*

**BHI** - Branch if Higher

Function:    **BHI**    goto specified address if C+Z=0  
              **LBHI**

**Description:**

Test the Zero and Carry flags in the CC register. If C+Z=1, then do nothing, and continue execution with the next instruction. If C+Z=0, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BHI or -32768 to +32767 for LBHI). Note that the PC points to the instruction following the BHI whilst it is being executed and that LBHI takes 6 cycles if the branch is taken and 5 if it isn't.

**Addressing Modes:**

	<b>BHI</b> ===		<b>LBHI</b> ====			
	code	cyc	byt	code	cyc	byt
Relative	\$22/34	3	2	\$10/16	5/6	4
				\$22/34		

**Condition Codes:**

	<b>E</b>	<b>F</b>	<b>H</b>	<b>I</b>	<b>Z</b>	<b>V</b>	<b>C</b>
<b>BHI</b>	*	*	*	*	*	*	*
<b>LBHI</b>	*	*	*	*	*	*	*

BHS - Branch on Higher or Same

Function:    BHS    goto specified address if C=0  
          LBHS

Description:

Test the Carry flag in the CC register. If C=1 then do nothing, and continue execution with the next instruction. If C=0, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BHS or -32768 to +32767 for LBHS). Note that the PC points to the instruction following the BHS whilst it is being executed and that LBHS takes 6 cycles if the branch is taken and 5 if it isn't. This instruction is exactly the same as BCC. The reason it is included is to make programs easier to read as this means Branch on Higher or the Same instead of Branch if Carry Clear.

Addressing Modes:

	BHS			LBHS			
	===			=====			
	code	cyc	byt	code	cyc	byt	
Relative	\$24/36	3	2	\$10/16	5/6	4	
				\$24/36			

Condition Codes:

	E	F	H	I	Z	V	C
BHS	*	*	*	*	*	*	*
LBHS	*	*	*	*	*	*	*



**BIT - Bit Test Accumulator**

Function:     BITA   A ^ M  
               BITB   B ^ M

**Description:**

Logically AND the contents of memory with the specified register. The result is not saved, and the contents of the accumulator are left undisturbed. It can be used to set the CC register without disturbing anything else.

**Addressing Modes:**

	BITA			BITB		
	code	cyc	byt	code	cyc	byt
Immediate	\$85/133	2	2	\$C5/197	2	2
Direct	\$95/149	4	2	\$D5/213	4	2
Extended	\$B5/181	5	3	\$F5/245	5	3
Indexed	\$A5/165	4+	2+	\$E5/229	4+	2+

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BITA	*	*	*	*	!	!	0	*
BITB	*	*	*	*	!	!	0	*

**BLE** - Branch on Less than or Equal to Zero

Function:    **BLE**    goto specified address if Z+N(XOR)V=1  
              **LBL**

**Description:**

Test the Zero, Negative and Overflow flags in the CC register. If Z+N(XOR)V=0, then do nothing, and continue execution with the next instruction. If Z+N(XOR)V=1, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BLE or -32768 to +32767 for LBLE). Note that the PC points to the instruction following the BLE whilst it is being executed and that LBLE takes 6 cycles if the branch is taken and 5 if it isn't.

**Addressing Modes:**

	<b>BLE</b>				<b>LBL</b>				
	===				====				
	code	cyc	byt		code	cyc	byt		
Relative	\$2F/47	3	2		\$10/16	5/6	4		
					\$2F/47				

**Condition Codes:**

	<b>E</b>	<b>F</b>	<b>H</b>	<b>I</b>	<b>Z</b>	<b>V</b>	<b>C</b>
<b>BLE</b>	*	*	*	*	*	*	*
<b>LBL</b>	*	*	*	*	*	*	*

BLO - Branch if Lower than

Function: BLO goto specified address if C=1  
LBLO

Description:

Test the Carry flag in the CC register. If C=0 then do nothing, and continue execution with the next instruction. If C=1, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BLO or -32768 to +32767 for LBLO). Note that the PC points to the instruction following the BLO whilst it is being executed and that LBLO takes 6 cycles if the branch is taken and 5 if it isn't. As with BHI and BCC this instruction is the same as BCS and is included for the same reason.

Addressing Modes:

	BLO				LBLO				
	===				====				
	code	cyc	byt		code	cyc	byt		
Relative	\$25/37	3	2		\$10/16	5/6	4		
					\$25/37				

Condition Codes:

	E	F	H	I	Z	V	C
BLO	*	*	*	*	*	*	*
LBLO	*	*	*	*	*	*	*

BLS - Branch if Lower than or the Same

Function: BLS goto specified address if Z+C=1  
LBS

Description:

Test the Zero and Carry flags in the CC register. If Z+C=0 then do nothing, and continue execution with the next instruction. If Z+C=1, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BLS or -32767 to +32768 for LBS). Note that the PC points to the instruction following the BLS whilst it is being executed and that LBS takes 6 cycles if the branch is taken and 5 if it isn't.

Addressing Modes:

	BLS			LBS		
	===			====		
	code	cyc	byt	code	cyc	byt
Relative	\$23/35	3	2	\$10/16	5/6	4
				\$23/35		

Condition Codes:

	E	F	H	I	Z	V	C
BLS	*	*	*	*	*	*	*
LBS	*	*	*	*	*	*	*

**BLT - Branch on Less Than Zero**

Function: BLT goto specified address if N(XOR)V=1  
LBLT

**Description:**

Test the Negative and overflow flags in the CC register. If N(XOR)V=0 then do nothing, and continue execution with the next instruction. If N(XOR)V=1, then branch to the current value of the PC plus the signed displacement (-128 to +127 for BLT or -32767 to +32768 for LBLT). Note that the PC points to the instruction following the BLT whilst it is being executed and that LBLT takes 6 cycles if the branch is taken and 5 if it isn't.

**Addressing Modes:**

	BLT			LBLT			
	===			====			
	code	cyc	byt	code	cyc	byt	
Relative	\$2D/45	3	2	\$10/16	5/6	4	\$2D/45

**Condition Codes:**

	E	F	H	I	Z	V	C
BLT	*	*	*	*	*	*	*
LBLT	*	*	*	*	*	*	*

## BMI - Branch on Minus

Function: BMI goto specified address if N=1  
LBMI

### Description:

Test the Negative flag in the CC register. If N=0, then do nothing, and continue execution with the next instruction. If N=1 then branch to the current value of the PC plus the signed displacement (-128 to +127 for BMI or -32768 to +32767 for LBMI). Note that the PC points to the instruction following the BMI whilst it is being executed.

### Addressing Modes:

	BMI		LBMI	
	code	cyc byt	code	cyc byt
Relative	\$2B/43	3 2	\$10/16	5+ 4
			\$2B/43	

### Condition Codes:

	E	F	H	I	N	Z	V	C
BMI	*	*	*	*	*	*	*	*
LBMI	*	*	*	*	*	*	*	*

**BNE - Branch on Not Equal to Zero**

Function:     BNE     goto specified address if Z=0  
               LBNE

**Description:**

Test the Zero flag in the CC register. If Z=1, then do nothing, and continue execution with the next instruction. If Z=0 then branch to the current value of the PC plus the signed displacement (-128 to +127 for BNE or -32768 to +32767 for LBNE). Note that the PC points to the instruction following the BNE whilst it is being executed.

**Addressing Modes:**

	BNE			LBNE		
	====			====		
	code	cyc	byt	code	cyc	byt
Relative	\$26/38	3	2	\$10/16	5+	4
				\$26/38		

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BNE	*	*	*	*	*	*	*	*
LBNE	*	*	*	*	*	*	*	*

BPL - Branch on Plus

Function: BPL goto specified address if N=0  
 LBPL

Description:

Test the Negative flag in the CC register. If N=1, then do nothing, and continue execution with the next instruction. If N=0 then branch to the current value of the PC plus the signed displacement (-128 to +127 for BPL or -32768 to +32767 for LBPL). Note that the PC points to the instruction following the BPL whilst it is being executed.

Addressing Modes:

	BPL		LBPL	
	===		====	
	code	cyc byt	code	cyc byt
Relative	\$2A/42	3 2	\$10/16	5+ 4
			\$2A/42	

Condition Codes:

	E	F	H	I	N	Z	V	C
BPL	*	*	*	*	*	*	*	*
LBPL	*	*	*	*	*	*	*	*



**BRA - Branch Always**

Function: BRA always goto specified address  
LBRA

**Description:**

Always branch to the current PC value plus the displacement (-128 to +127 for BRA or =32768 to +32767 for LBRA). Note that the PC points to the instruction following the BMI whilst it is being executed.

**Addressing Modes:**

	BRA		LBRA	
	code	cyc byt	code	cyc byt
Relative	\$20/32	3 2	\$10/16	5 4
			\$20/32	

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BRA	*	*	*	*	*	*	*	*
LBRA	*	*	*	*	*	*	*	*

**BRN - Branch Never**

Function: BRN never goto specified address  
LBRN

**Description:**

Never branch. This instruction is often used instead of NOP's because it takes fewer bytes to produce a time delay than do the equivalent number of NOP's.

**Addressing Modes:**

	BRN ===			LBRN ====		
	code	cyc	byt	code	cyc	byt
Relative	\$21/33	3	2	\$10/16	5	4
				\$21/33		

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BRN	*	*	*	*	*	*	*	*
LBRN	*	*	*	*	*	*	*	*

**BSR - Branch to Subroutine**

Function:    BSR    S   ← S - 1 ; (S)   ← PC-low  
                       S   ← S - 1 ; (S)   ← PC-high  
                       PC   ← PC + M

**Description:**

Branch to the current value of the PC plus the signed displacement (-128 to 127 for BSR or -32768 to +32767 for LBSR), having first pushed the value of the PC onto the S stack. Note that the PC points to the instruction following the BSR whilst it is being executed.

**Addressing Modes:**

	BSR			LBSR		
	===			====		
	code	cyc	byt	code	cyc	byt
Relative	\$8D/141	3	2	\$10/16	5+	4
				\$17/23		

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BSR	*	*	*	*	*	*	*	*
LBSR	*	*	*	*	*	*	*	*

**BVC - Branch on Overflow Clear**

Function: BVC goto specified address if V=0  
 LBVC

**Description:**

Test the Overflow flag in the CC register. If V=1, then do nothing, and continue execution with the next instruction. If V=0 then branch to the current value of the PC plus the signed displacement (-128 to +127 for BVC or -32768 to +32767 for LBVC). Note that the PC points to the instruction following the BVC whilst it is being executed.

**Addressing Modes:**

	BVC		LBVC	
	===		====	
	code	cyc byt	code	cyc byt
Relative	\$28/40	3 2	\$10/16	5+ 4
			\$28/40	

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BVC	*	*	*	*	*	*	*	*
LBVC	*	*	*	*	*	*	*	*

**BVS - Branch on Overflow Set**

**Function:** BVS goto specified address if V=1  
LBVS

**Description:**

Test the Overflow flag in the CC register. If V=0, then do nothing, and continue execution with the next instruction. If V=1 then branch to the current value of the PC plus the signed displacement (-128 to +127 for BVS or -32768 to +32767 for LBVS). Note that the PC points to the instruction following the BVS whilst it is being executed.

**Addressing Modes:**

	BVS			LBVS		
	code	cyc	byt	code	cyc	byt
Relative	\$29/41	3	2	\$10/16	5+	4
				\$29/41		

**Condition Codes:**

	E	F	H	I	N	Z	V	C
BVS	*	*	*	*	*	*	*	*
LBVS	*	*	*	*	*	*	*	*

**CLR - Clear Accumulator or Memory**

Function:    CLRA    A ← 0  
               CLRB    B ← 0  
               CLR     M ← 0

Description:  
 Clear the specified accumulator or memory location to zero.

**Addressing Modes:**

	CLRA			CLRB			CLR		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$4F/79	2	1	\$5F/95	2	1			
Direct							\$0F/15	6	2
Extended							\$7F/127	7	3
Indexed							\$6F/111	6+	2+

**Condition Codes:**

	E	F	H	I	N	Z	V	C
CLRA	*	*	*	*	0	1	0	0
CLRB	*	*	*	*	0	1	0	0
CLR	*	*	*	*	0	1	0	0

**CMP - Compare to Register**

Function:   CMPA   A ← M  
               CMPB   B ← M  
               CMPD   D ← M:M+1  
               CMPS   S ← M:M+1  
               CMPU   U ← M:M+1  
               CMPX   X ← M:M+1  
               CMPY   Y ← M:M+1

Description:  
 Subtract the contents of memory from the specified register.  
 The contents of the accumulator are left undisturbed, but the  
 CC register will be affected.

Addressing modes:

	CMPA ====			CMPB ====			CMPD ====		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Immediate	\$81/129	2	2	\$C1/193	2	2	\$10/16	5	4
Direct	\$91/145	4	5	\$D1/209	4	2	\$10/16	7	3
Extended	\$B1/177	5	3	\$F1/241	5	3	\$10/16	8	4
Indexed	\$A1/161	4+	2+	\$E1/225	4+	2+	\$10/16	7+	3+
							\$83/131		
							\$93/147		
							\$A3/163		
							\$A3/163		

	CMPS ====			CMPU ====			CMPX ====		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Immediate	\$11/17	5	4	\$11/17	5	4	\$8C/140	4	3
Direct	\$11/17	7	3	\$11/17	7	3	\$9C/156	6	2
Extended	\$11/17	8	4	\$11/17	8	4	\$BC/188	7	3
Indexed	\$11/17	7+	3+	\$11/17	7+	3+	\$AC/172	6+	2+
	\$8C/140			\$83/131					
	\$9C/156			\$93/147					
	\$BC/188			\$B3/179					
	\$AC/172			\$A3/163					

	CMPY		
	====		
	code	cyc	byt
Immediate	\$10/16	5	4
	\$8C/140		
Direct	\$10/16	7	3
	\$9C/156		
Extended	\$10/16	8	4
	\$BC/188		
Indexed	\$10/16	7+	3+
	\$AC/172		

Condition Codes:

	E	F	H	I	N	Z	V	C
CMPA	*	*	!	*	!	!	!	!
CMPB	*	*	!	*	!	!	!	!
CMPD	*	*	*	*	!	!	!	!
CMPS	*	*	*	*	!	!	!	!
CMPU	*	*	*	*	!	!	!	!
CMPX	*	*	*	*	!	!	!	!
CMPY	*	*	*	*	!	!	!	!



COM - Complement

Function:    COMA   A ←  $\bar{A}$   
               COMB   B ←  $\bar{B}$   
               COM    M ←  $\bar{M}$

Description:

The operand byte is replaced by it's logical or 1's-complement, i.e. all 0's are made 1's and vice-versa.

Addressing Modes:

	COMA =====	COMB =====	COM =====
	code cyc byt	code cyc byt	code cyc byt
Inherent	\$43/67 2 1	\$53/83 2 1	
Direct			\$03/03 6 2
Extended			\$73/115 7 3
Indexed			\$63/99 6+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
COMA	*	*	*	*	!	!	0	1
COMB	*	*	*	*	!	!	0	1
COM	*	*	*	*	!	!	0	1

CWAI - Clear CC and wait for interrupt

Function: CWAI #N CC ← CC+N  
 E ← 1  
 S ← S-1 ; (S) ← PC-low  
 S ← S-1 ; (S) ← PC-high  
 S ← S-1 ; (S) ← U-low  
 S ← S-1 ; (S) ← U-high  
 S ← S-1 ; (S) ← Y-low  
 S ← S-1 ; (S) ← Y-high  
 S ← S-1 ; (S) ← X-low  
 S ← S-1 ; (S) ← X-high  
 S ← S-1 ; (S) ← DP  
 S ← S-1 ; (S) ← B  
 S ← S-1 ; (S) ← A  
 S ← S-1 ; (S) ← CC

Desciption:

The operand byte is logically ANDed with the condition code register. This action may clear specific bits, e.g., the interrupt masks. The E bit is then set and the entire processor state is saved on the S stack. The processor then waits for an interrupt to occur, an RTI instruction will restore the entire processor state upon finding the E bit set in the recovered CC register.

Addressing modes:

CWAI  
 ====  
 code cyc byt

Immediate \$3C/60 20 2

Condition Codes:

	E	F	H	I	N	Z	V	C
CWAI	1	?	?	?	?	?	?	?

(Changed according to operand)

## DAA - Decimal Addition Adjust

Function: DAA

A ← A + Correction

Correction:

Least Significant Nybble:

6: if H = 1, or if LSN > 9

0: otherwise

Most Significant Nybble:

6: if C = 1, or MSN > 9, or MSN > 8  
and LSN > 9

0: otherwise

### Description:

The appropriate correction factor is computed based on the values of the two nybbles in the A register and the CC bits. It is then added to A. This instruction is used after the addition of two BCD numbers to assure a proper BCD result. The Carry bit generated by this instruction is the correct carry of BCD addition.

### Addressing Modes:

DAA

===

code cyc byt

Inherent \$19/25 2 1

### Condition Codes:

	E	F	H	I	N	Z	V	C
DAA	*	*	*	*	!	!	0	!

DEC - Decrement

Function: DECA A ← A-1  
 DECB B ← B-1  
 DEC M ← M-1

Description:

One is subtracted from either one of the accumulators or the specified memory location. Note that the carry bit is not affected.

Addressing Modes:

	DECA			DECB			DEC
	====			====			====
	code	cyc	byt	code	cyc	byt	code
Inherent	\$4A/58	2	1	\$5A/90	2	1	
Direct							\$0A/10 6 2
Extended							\$7A/122 7 3
Indexed							\$6A/112 6+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
DECA	*	*	*	*	!	!	!	*

set only if  
 original operand was \$80,  
 cleared otherwise.

**EOR - Exclusive OR**

Function: EORA A ← A (XOR) M  
EORB B ← B (XOR) M

**Description:**

The operand is logically Exclusive ORed into the specified accumulator. EORing produces a 1 if either of the operands are 1 and a 0 if both the operands are the same.

**Addressing Modes:**

	EORA			EORB		
	====			====		
	code	cyc	byt	code	cyc	byt
Immediate	\$88/136	2	2	\$C8/200	2	2
Direct	\$98/152	4	2	\$D8/216	4	2
Extended	\$B8/184	5	3	\$F8/248	5	3
Indexed	\$A8/168	4+	2+	\$E8/232	4+	2+

**Condition Codes:**

	E	F	H	I	N	Z	V	C
EROA	*	*	*	*	!	!	0	*
EORB	*	*	*	*	!	!	0	*

EXG - Exchange Registers

Function: EXG R1,R2 R1 ↔ R2

Description:

The register's values specified by the postbyte of the instruction are exchanged. The low and high nibbles of the postbyte specify the registers to be exchanged in the following way:

0000 = 0 = D	1000 = 8 = A
0001 = 1 = X	1001 = 9 = B
0010 = 2 = Y	1010 = A = CC
0011 = 3 = U	1011 = B = DP
0100 = 4 = S	6, 7, C, D, E, F, = undefined
0101 = 5 = PC	

Only registers of like size can be exchanged.

Addressing Modes:

EXG  
===  
code cyc byt

Inherent \$1E/30 7 2

Condition Codes:

	E	F	H	I	N	Z	V	C
EXG	*	*	*	*	*	*	*	*

(No change unless one of the registers is CC)

**INC - Increment**

Function: INCA A ← A + 1  
 INCB B ← B + 1  
 INC M ← M + 1

Description: One is added to the operand. Note that the carry bit is not affected.

Addressing Modes:

	INCA			INCB			INC		
	====			====			===		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$4C/76	2	1	\$5C/92	2	1			
Direct							\$0C/12	6	2
Extended							\$7C/124	7	3
Indexed							\$6C/108	6+	2+

Condition Codes:

	E	F	H	I	N	Z	V	C
INCA	*	*	*	*	!	!	!	*
INCB	*	*	*	*	!	!	!	*
INC	*	*	*	*	!	!	!	*

## JMP - Jump

Function: JMP PC + ADDR

### Description:

The value of the operand is transferred to the PC, and program execution continues from that address.

### Addressing Modes:

	JMP			
	===			
	code	cyc	byt	
Direct	\$0E/14	3	2	
Extended	\$7E/126	4	3	
Indexed	\$6E/110	3+	2+	

### Condition Codes:

	E	F	H	I	N	Z	V	C
JMP	*	*	*	*	*	*	*	*



## JSR - Jump to Subroutine

Function: JSR S ← S-1 ; (S) ← PC-low  
S ← S-1 ; (S) ← PC-high  
PC ← ADDR

### Description:

The PC is pushed onto the S stack. The value of the operand is put into the PC and execution continues. An RTS (return from subroutine) will return control to the instruction following the JSR instruction.

### Addressing Modes:

	JSR			
	===			
	code	cyc	byt	
Direct	\$9D/157	7	2	
Extended	\$BD/189	8	3	
Indexed	\$AD/173	7+	2+	

### Condition Codes:

	E	F	H	I	N	Z	V	C
JSR	*	*	*	*	*	*	*	*

LD - Load Register From Memory

Function: LDA A ← M  
 LDB B ← M  
 LDD D ← M:M+1  
 LDS S ← M:M+1  
 LDU U ← M:M+1  
 LDX X ← M:M+1  
 LDY Y ← M:M+1

Description:

The operand is loaded into the specified register.

Addressing Modes:

	LDA ===			LDB ===			LDD ===		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Immediate	\$86/134	2	2	\$C6/198	2	2	\$CC/204	3	3
Direct	\$96/150	4	2	\$D6/214	4	2	\$DC/220	5	2
Extended	\$B6/182	5	3	\$F6/246	5	3	\$FC/252	6	3
Indexed	\$A6/166	4+	2+	\$E6/230	4+	2+	\$EC/236	4+	2+

	LDS ===			LDU ===			LDX ===		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Immediate	\$10/16	4	4	\$CE/206	3	3	\$8E/142	3	3
Direct	\$CE/206								
	\$10/16	6	3	\$DE/222	5	2	\$9E/158	5	2
Extended	\$DE/222								
	\$10/16	7	4	\$FE/254	6	3	\$BE/190	6	3
Indexed	\$FE/254								
	\$10/16	6+	3+	\$EE/238	5+	2+	\$AE/174	5+	2+
	\$EE/238								

LDY

===

code cyc byt

Immediate	\$10/16	4	4
	\$8E/142		
Direct	\$10/16	6	3
	\$9E/158		
Extended	\$10/16	7	4
	\$BE/190		
Indexed	\$10/16	6+	3+
	\$AE/174		

Condition Codes:

	E	F	H	I	N	Z	V	C
LDA	*	*	*	*	!	!	0	*
LDB	*	*	*	*	!	!	0	*
LDD	*	*	*	*	!	!	0	*
LDS	*	*	*	*	!	!	0	*
LDU	*	*	*	*	!	!	0	*
LDX	*	*	*	*	!	!	0	*
LDY	*	*	*	*	!	!	0	*

LEA - Load Effective Address

Function: LEAS S ← ADDR  
 LEAU U ← ADDR  
 LEAX X ← ADDR  
 LEAY Y ← ADDR

Description:

The specified register is loaded with the value of the effective address which would normally be used to fetch the operand instead of the actual operand. The only addressing mode available is Indexed.

Addressing Modes:

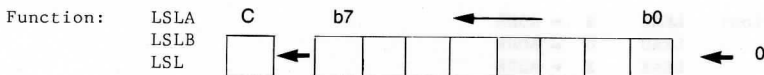
	LEAS ====	LEAU ====	LEAX ====
	code cyc byt	code cyc byt	code cyc byt
Indexed	\$32/50 4+ 2+	\$33/51 4+ 2+	\$30/48 4+ 2+

	LEAY ====
	code cyc byt
Indexed	\$31/49 4+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
LEAS	*	*	*	*	*	*	*	*
LEAU	*	*	*	*	*	*	*	*
LEAX	*	*	*	*	*	!	*	*
LEAY	*	*	*	*	*	!	*	*

## LSL - Logical Shift Left



### Description:

Shift the contents of memory or one of the accumulators to the left one bit position, with bit 7 going into the Carry flag, and a 0 going into bit 0. The result is stored back in the specified memory location or accumulator. Note that this instruction is exactly the same as ASL. This instruction is included to be consistent (i.e. two of both types of shifts).

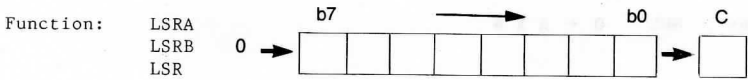
### Addressing Modes:

	LSLA			LSLB			LSL		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$48/72	2	1	\$58/88	2	1			
Direct							\$08/8	6	2
Extended							\$78/120	7	3
Indexed							\$68/104	6+	2+

### Condition Codes:

	E	F	H	I	N	Z	V	C
ASLA	*	*	?	*	!	!	!	!
ASLB	*	*	?	*	!	!	!	!
ASL	*	*	?	*	!	!	!	!

LSR - Logical Shift Right



Description:  
 Shift the contents of memory, or either of the accumulators to the right one bit position, with bit 0 going into the Carry bit, and a 0 going into bit 0. The result is stored back into the specified memory location, or accumulator.

Addressing Modes:

	LSRA ====		LSRB ====		LSR ===	
	code	cyc byt	code	cyc byt	code	cyc byt
Inherent	\$44/68	2 1	\$54/84	2 1		
Direct					\$04/4	6 2
Extended					\$74/116	7 3
Indexed					\$64/100	6+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
LSRA	*	*	*	*	0	!	*	!
LSRB	*	*	*	*	0	!	*	!
LSR	*	*	*	*	0	!	*	!

**MUL - Multiply**

Function: MUL D ← A X B



**Description:**

The two accumulators (unsigned) are multiplied together and the result (unsigned) is stored in the D register. The original values of A and B are destroyed.

**Addressing Modes:**

MUL  
===  
code cyc byt

Inherent \$3D/61 11 1

**Condition Codes:**

	E	F	H	I	N	Z	V	C
MUL	*	*	*	*	*	!	*	*

↖ Set if  
bit 7 of B set  
in the result.

NEG - Negate

Function:    NEGA    A ← 0 - A  
               NEGB    B ← 0 - B  
               NEG     M ← 0 - M

Description:

The number in memory or one of the accumulators is replaced with it's two's-complment.

Addressing Modes:

	NEGA ====	NEGB ====	NEG ===
	code cyc byt	code cyc byt	code cyc byt
Inherent	\$40/64 2 1	\$50/80 2 1	
Direct			\$00/0 6 2
Extended			\$70/112 7 3
Indexed			\$60/96 6+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
NEGA	*	*	?	*	!	!	!	!
NEGB	*	*	?	*	!	!	!	!
NEG	*	*	?	*	!	!	!	!



**NOP - No Operation**

Function:    NOP    Nothing

Description:  
Nothing happens, no registers are affected, no memory is accessed. Can be used for creating a delay or room for expansion in a program

Addressing Modes:

          NOP  
          ===  
          code cyc byt  
  
Inherent \$12/18 2 1

Condition Codes:

	E	F	H	I	N	Z	V	C
NOP	*	*	*	*	*	*	*	*

## OR - OR Memory Into Register

Function:   ORA    A ← A (OR) M  
           ORB    B ← B (OR) M  
           ORCC   CC ← CC (OR) M

### Description:

The specified accumulator and the memory operand are logically ORed together, and the result is stored in the same register. The ORCC instruction can use Immediate addressing only, and can be used to set an interrupt mask or status.

### Addressing Modes:

	ORA ===		ORB ===		ORCC ====	
	code	cyc	byt	code	cyc	byt
Immediate	\$8A/138	2	2	\$CA/202	2	2
Direct	\$9A/154	4	2	\$DA/218	4	2
Extended	\$BA/186	5	3	\$FA/250	5	3
Indexed	\$AA/170	4+	2+	\$EA/234	4+	2+

### Condition Codes:

	E	F	H	I	N	Z	V	C
ORA	*	*	*	*	!	!	0	*
ORB	*	*	*	*	!	!	0	*
ORCC	?	?	?	?	?	?	?	?

(Changed according to operand)

PSH - Push Registers onto either Stack

Function: PSHS if b7 of postbyte set: S ← S - 1 ;  
 (S) ← PC-low  
 S ← S - 1 ;  
 (S) ← PC-high  
 if b6 of postbyte set: S ← S - 1 ;  
 (S) ← U-low  
 S ← S - 1 ;  
 (S) ← U-high  
 if b5 of postbyte set: S ← S - 1 ;  
 (S) ← Y-low  
 S ← S - 1 ;  
 (S) ← Y-high  
 if b4 of postbyte set: S ← S - 1 ;  
 (S) ← X-low  
 S ← S - 1 ;  
 (S) ← X-high  
 if b3 of postbyte set: S ← S - 1 ;  
 (S) ← DP  
 if b2 of postbyte set: S ← S - 1 ;  
 (S) ← B  
 if b1 of postbyte set: S ← S - 1 ;  
 (S) ← A  
 if b0 of postbyte set: S ← S - 1 ;  
 (S) ← CC  
 PSHU Same as above except registers are  
 pushed onto the U stack.

Description:

Any, all, or none of the registers are pushed onto either the U or S stack. The postbyte is determined by which registers are to be pushed onto the stack. The postbyte has the following structure:

b7	b6	b5	b4	b3	b2	b1	b0	
PC	U	Y	X	DP	B	A	CC	
								push order --- →

One register may be pushed with an autodecrement store.  
 Example: STY ,--S pushes Y, but also changes CC register bits.

Addressing Modes:

	PSHS			PSHU		
	====			====		
	code	cyc	byt	code	cyc	byt
Immediate	\$34/52	5+	2	\$36/54	5+	2

Condition Codes:

	E	F	H	I	N	Z	V	C
PSHS	*	*	*	*	*	*	*	*
PSHU	*	*	*	*	*	*	*	*

## PUL - Pull Registers from either Stack

Function: PULS if b0 of postbyte set: CC ← (S) ;  
S ← S + 1  
if b1 of postbyte set: A ← (S) ;  
S ← S + 1  
if b2 of postbyte set: B ← (S) ;  
S ← S + 1  
if b3 of postbyte set: DP ← (S) ;  
S ← S + 1  
if b4 of postbyte set: X-high (- (S) ;  
S ← S + 1  
X-low (- (S) ;  
S ← S + 1  
if b5 of postbyte set: Y-high (- (S) ;  
S ← S + 1  
Y-low (- (S) ;  
S ← S + 1  
if b6 of postbyte set: U-high (- (S) ;  
S ← S + 1  
U-low (- (S) ;  
S ← S + 1  
if b7 of postbyte set: PC-high (- (S) ;  
S ← S + 1  
PC-low (- (S) ;  
S ← S + 1  
PULU Same as above except that registers are  
pulled from the U stack.

### Description:

Any, all or none of the registers are pulled from either the S or the U stack. The postbyte is determined by which registers are to be pulled from the stack. The postbyte has the following structure:

b7 b6 b5 b4 b3 b2 b1 b0

PC U Y X DP B A CC

← --- pull order

One register may be pulled with an auto-increment load.

Example: LDY ,S++ pulls Y, but also changes the CC register bits.

Addressing Modes:

	PULS			PULU		
	====			====		
	code	cyc	byt	code	cyc	byt
Immediate	\$35/53	5+	2	\$37/55	5+	2

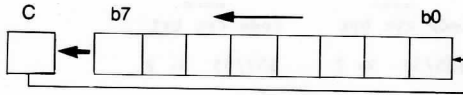
Condition Codes:

	E	F	H	I	N	Z	V	C
PULS	*	*	*	*	*	*	*	*
PULU	*	*	*	*	*	*	*	*

(No change unless CC pulled)

ROL - Rotate Left

Function: ROLA  
 ROLB  
 ROL



Description:

All of the bits in the operand are rotated left one bit position, with the Carry bit going into bit 0, and bit 7 going into the Carry bit.

Addressing Modes:

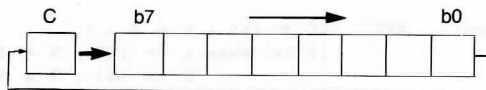
	ROLA ====	ROLB ====	ROL ===
	code cyc byt	code cyc byt	code cyc byt
Inherent	\$46/70 2 1	\$56/86 2 1	
Direct			\$09/9 6 2
Extended			\$79/121 7 3
Indexed			\$69/105 6+ 2+

Condition Codes:

	E	F	H	I	N	Z	V	C
ROLA	*	*	*	*	!	!	!	!
ROLB	*	*	*	*	!	!	!	!
ROL	*	*	*	*	!	!	!	!

ROR - Rotate Right

Function: RORA  
RORB  
ROR



Description:

All the bits of the operand are shifted right one bit position, with the Carry bit going into bit 7, and bit 0 going into the Carry bit.

Addressing Modes:

	RORA ====			RORB ====			ROR ====		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$46/20	2	1	\$56/36	2	1			
Direct							\$06/6	6	2
Extended							\$76/118	7	3
Indexed							\$66/102	6+	2+

Condition Codes:

	E	F	H	I	N	Z	V	C
RORA	*	*	*	*	!	!	*	!
RORB	*	*	*	*	!	!	*	!
ROR	*	*	*	*	!	!	*	!



RTI - Return from Interrupt

Function: RTI CC ← (S) ; S ← S + 1  
 if E=1 then: A ← (S) ; S ← S + 1  
 B ← (S) ; S ← S + 1  
 DP ← (S) ; S ← S + 1  
 X-high ← (S) ; S ← S + 1  
 X-low ← (S) ; S ← S + 1  
 Y-high ← (S) ; S ← S + 1  
 Y-low ← (S) ; S ← S + 1  
 U-high ← (S) ; S ← S + 1  
 U-low ← (S) ; S ← S + 1  
 PC-high ← (S) ; S ← S + 1  
 PC-low ← (S) ; S ← S + 1

Description:

The CC register is pulled from the S stack. If the E flag is set in the CC register then the entire set of registers is pulled from the stack. This instruction reverses the effects of an interrupt and should be placed at the end of an interrupt handling routine.

Addressing Modes:

RTI  
 ===  
 code cyc byt  
 Inherent \$3B/59 6/15 1

Condition Codes:

	E	F	H	I	N	Z	V	C
RTI	?	?	?	?	?	?	?	?

(Pulled from Stack)

RTS - Return from Subroutine

Function:     RTS     PC-high  $\leftarrow$  (S) ; S  $\leftarrow$  S + 1  
                      PC-low   $\leftarrow$  (S) ; S  $\leftarrow$  S + 1

Description:

The PC is pulled from the S stack. This instruction reverses the effects of the BSR and JSR instructions and should be placed at the end of a subroutine.

Addressing Modes:

      RTS  
      ===  
code cyc byt

Inherent \$39/57 5 1

Condition Codes:

	E	F	H	I	N	Z	V	C
RTS	*	*	*	*	*	*	*	*

SBC - Subtract with Borrow

Function:    SBCA    A ← A - M - C  
               SBCB    B ← B - M - C

Description:

The operand and the Carry bit are subtracted from the specified accumulator. The resulting Carry bit is a borrow and is set to the complement of the carry of the internal addition.

Addressing Modes:

	SBCA			SUCB		
	=====			=====		
	code	cyc	byt	code	cyc	byt
Immediate	\$82/130	2	2	\$C2/194	2	2
Direct	\$92/146	4	2	\$D2/210	4	2
Extended	\$B2/178	5	3	\$F2/242	5	3
Indexed	\$A2/162	4+	2+	\$E2/226	4+	2+

Condition Codes:

	E	F	H	I	N	Z	V	C
SBCA	*	*	?	*	!	!	!	!
SBCB	*	*	?	*	!	!	!	!

SEX - Sign Extend

Function:   SEX    If b7 of B = 1 then: A ← \$FF  
  else: A ← \$0

Discription:

The 8-bit two's complement value in accumulator B is sign extended to a 16-bit value in accumulator D. The original value of A is lost.

Addressing Modes:

      SEX  
      ===  
code cyc byt

Inherent \$1D/29 2 1

Condition Codes:

	E	F	H	I	N	Z	V	C
SEX	*	*	*	*	!	!	0	*

## ST - Store Register into Memory

Function:    STA    M ← A  
              STB    M ← B  
              STD    M:M+1 ← D  
              STU    M:M+1 ← S  
              STU    M:M+1 ← U  
              STX    M:M+1 ← X  
              STY    M:M+1 ← Y

### Description:

The contents of the specified register are stored at the address specified by the operand.

### Addressing Modes:

	STA ===	STB ===	STD ===
	code cyc byt	code cyc byt	code cyc byt
Direct	\$97/151 4 2	\$D7/215 4 2	\$DD/221 5 2
Extended	\$B7/183 5 3	\$F7/247 5 3	\$FD/253 6 3
Indexed	\$A7/167 4+ 2+	\$E7/231 4+ 2+	\$ED/237 5+ 2+

	STS ===	STU ===	STX ===
	code cyc byt	code cyc byt	code cyc byt
Direct	\$10/16 6 3 \$DF/223	\$DF/223 5 2	\$9F/159 5 2
Extended	\$10/16 7 4 \$FF/255	\$FF/255 6 3	\$BF/191 6 3
Indexed	\$10/16 6+ 3+ \$EF/239	\$EF/239 5+ 2+	\$AF/175 5+ 2+

STY  
 ===  
 code cyc byt

Direct    \$10/16 6 3  
           \$9F/159  
 Extended \$10/16 7 4  
           \$BF/191  
 Indexed  \$10/16 6+ 3+  
           \$AF/175

Condition Codes:

	E	F	H	I	N	Z	V	C
STA	*	*	*	*	!	!	0	*
STB	*	*	*	*	!	!	0	*
STD	*	*	*	*	!	!	0	*
STS	*	*	*	*	!	!	0	*
STU	*	*	*	*	!	!	0	*
STX	*	*	*	*	!	!	0	*
STY	*	*	*	*	!	!	0	*

**SUB** - Subtract from Register

Function:   SUBA   A ← A - M  
               SUBB   B ← B - M  
               SUBD   D ← D - M:M+1

**Description:**

The operand is subtracted from the specified accumulator. The Carry bit is a borrow bit and is set to the complement of the carry of the internal binary addition.

**Addressing Modes:**

	SUBA =====	SUBB =====	SUBD =====
	code cyc byt	code cyc byt	code cyc byt
Immediate	\$80/128 2 2	\$C0/192 2 2	\$83/131 4 3
Direct	\$90/144 4 2	\$D0/208 4 2	\$93/147 6 2
Extended	\$B0/176 5 3	\$F0/240 5 3	\$B3/179 7 3
Indexed	\$A0/160 4+ 2+	\$E0/224 4+ 2+	\$A3/163 6+ 2+

**Condition Codes:**

	E	F	H	I	N	Z	V	C
SUBA	*	*	?	*	!	!	!	!
SUBB	*	*	?	*	!	!	!	!
SUBD	*	*	*	*	!	!	!	!

SWI - Software Interrupt

Function: SWI E ← 1  
 S ← S - 1 ; (S) ← PC-low  
 S ← S - 1 ; (S) ← PC-high  
 S ← S - 1 ; (S) ← U-low  
 S ← S - 1 ; (S) ← U-high  
 S ← S - 1 ; (S) ← Y-low  
 S ← S - 1 ; (S) ← Y-high  
 S ← S - 1 ; (S) ← X-low  
 S ← S - 1 ; (S) ← X-high  
 S ← S - 1 ; (S) ← DP  
 S ← S - 1 ; (S) ← B  
 S ← S - 1 ; (S) ← A  
 S ← S - 1 ; (S) ← CC  
 I ← 1 ; F (- 1  
 PC-high ← FFFA ; PC-low ← FFFB

Description:

The entire machine state is pushed onto the S stack. Program control is transferred via the software interrupt 1 vector. Fast and normal interrupts are disabled by setting the I and F flags in the CC register AFTER it has been pushed onto the stack, therefore when an RTI instruction pulls the registers off the stack the interrupt flags are returned to their original status.

Addressing Modes:

SWI  
 ===  
 code cyc byt

Inherent \$3F/63 19 1

Condition Codes:

	E	F	H	I	N	Z	V	C
SWI	1	1	*	1	*	*	*	*

↑ These are for the duration of the interrupt handling routine only.



SWI2 - Software Interrupt 2

Function: SWI2 E ← 1  
 S ← S - 1 ; (S) ← PC-low  
 S ← S - 1 ; (S) ← PC-high  
 S ← S - 1 ; (S) ← U-low  
 S ← S - 1 ; (S) ← U-high  
 S ← S - 1 ; (S) ← Y-low  
 S ← S - 1 ; (S) ← Y-high  
 S ← S - 1 ; (S) ← X-low  
 S ← S - 1 ; (S) ← X-high  
 S ← S - 1 ; (S) ← DP  
 S ← S - 1 ; (S) ← B  
 S ← S - 1 ; (S) ← A  
 S ← S - 1 ; (S) ← CC  
 PC-high ← FFF4 ; PC-low ← FFF5

Description:

The entire machine state is pushed onto the S stack. Program control is transferred via the software interrupt 2 vector.

Addressing Modes:

SWI2

===

code cyc byt

Inherent \$10/16 20 2  
 \$3F/63

Condition Codes:

	E	F	H	I	N	Z	V	C
SWI	1	*	*	*	*	*	*	*

SWI3 - Software Interrupt 3

Function: SWI E ← 1  
 S ← S - 1 ; (S) ← PC-low  
 S ← S - 1 ; (S) ← PC-high  
 S ← S - 1 ; (S) ← U-low  
 S ← S - 1 ; (S) ← U\_high  
 S ← S - 1 ; (S) ← Y-low  
 S ← S - 1 ; (S) ← Y-high  
 S ← S - 1 ; (S) ← X-low  
 S ← S - 1 ; (S) ← X-high  
 S ← S - 1 ; (S) ← DP  
 S ← S - 1 ; (S) ← B  
 S ← S - 1 ; (S) ← A  
 S ← S - 1 ; (S) ← CC  
 PC-high ← FFF2 ; PC-low ← FFF3

Description:

The entire machine state is pushed onto the S stack. Program control is transferred via the software interrupt 3 vector.

Addressing Modes:

SWI  
 ===  
 code cyc byt  
 Inherent \$10/16 20 2  
 \$3F/63

Condition Codes:

	E	F	H	I	N	Z	V	C
SWI	1	*	*	*	*	*	*	*

**SYNC - Synchronize to External Event**

Function      SYNC      Suspend processor

**Discription**

The processor stops and waits for an interrupt. If the interrupt is masked (it's flag set in the CC register) or does not last 3 cycles, then the processor continues execution with the instruction following the SYNC instruction. If the interrupt is enabled and lasts for more than 3 cycles, then the normal routine for that interrupt is performed, the return address being that of the instruction after the SYNC instruction. This instruction can be used to synchronize the processor with high speed, critical events, such as reading data from a disk drive.

Addressing Modes:

SYNC

====

code cyc byt

Inherent \$13/19 2+ 1

**Condition Codes:**

	E	F	H	I	N	Z	V	C
SYNC	*	*	*	*	*	*	*	*

**TFR** - Transfer Register to Register

Function:    **TFR**    R2 ← R1

**Description:**

The contents of the register specified by the high nybble of the postbyte are transferred to the register specified by the low nybble of the postbyte. The nybbles of postbyte specify the registers in the following way:

0000 = 0 = D	1000 = 8 = A
0001 = 1 = X	1001 = 9 = B
0010 = 2 = Y	1010 = A = CC
0011 = 3 = U	1011 = B = DP
0100 = 4 = S	6,7,C,D,E,F: Undefined
0101 = 5 = PC	

Only registers of the same size can be transferred. The contents of the register being transferred from is unchanged but the old contents of the register being transferred to is lost.

**Addressing Modes:**

**TFR**  
      ===  
code cyc byt

Immediate \$1F/31 7 2

**Condition Codes:**

	E	F	H	I	N	Z	V	C
<b>TFR</b>	*	*	*	*	*	*	*	*

(unless the CC register has something transferred into it)

TST - Test

Function: TSTA A - 0  
 TSTB B - 0  
 TST M - 0

Description:

The Z and N bits are set/reset according to the value in the accumulators or memory. The V bit is cleared and nothing else is affected.

Addressing Modes:

	TSTA =====			TSTB =====			TST ====		
	code	cyc	byt	code	cyc	byt	code	cyc	byt
Inherent	\$4D/77	2	1	\$5D/93	2	1			
Direct							\$0D/13	6	2
Extended							\$7D/125	7	3
Indexed							\$6D/109	6+	2+

Condition Codes:

	E	F	H	I	N	Z	V	C
TST	*	*	*	*	!	!	0	*

## CHAPTER 14

# Demonstration Programs

### Introduction

The aim of this chapter is to demonstrate each of the types of instructions that the 6809 has. These instructions have been presented to you already in chapter 13 but this chapter lets you get your hands dirty by actually using programs that use these instructions. These programs are already written for you and each one demonstrates one or more machine instructions.

Before going onto the programs there are three points which must be discussed:

#### 1. Choosing an area of RAM

A user-written machine code program must be protected from the roving BASIC interpreter (which likes to think it owns all the RAM), by CLEARing a section of memory at the top of RAM and making sure that all your machine code programs stay above this point. This will prevent the battle of memory which usually ends up with both BASIC and machine code programs destroyed.

In this chapter all BASIC programs CLEAR above &H4000 and all machine code programs start at &H4000 and continue up out of harms way.

Note that in BASIC hexadecimal numbers are denoted by a leading '&H' where as in assembler language they are denoted as '\$'.

2. Entering the machine code program.

There is no provision on the DRAGON for entering numbers directly into memory other than POKE (and CLOADM but that's not very direct). The POKE command can use any numeric expression including decimal, hexadecimal, and octal constants.

The recommended way to enter machine code programs (if you do not own an assembler) is to use the following 'loader' routine. Even if you do have an assembler it is still good practice to use the 'loader' routine.

```
00 SMEM = &H4000
10 DEF USR1 = &H4000
20 CLEAR 200, &H3FFF
30 READ XX$: IF XX$ = "***" THEN 70
40 POKE SMEM, VAL ( "&H" + XX$ )
50 SMEM = SMEM + 1 : IF SMEM < &H8000 THEN 30
60 PRINT "ERROR! - OUT OF MEMORY"
70 PRINT "FINISHED LOADING"
```

The 'loader' will read hexadecimal numbers and store them into memory locations &H4000 onwards until it comes across a pair of asterisks ( '\*\*' ).

For example:

```
100 DATA 00,01,39,**
will load &H4000 with &H00, &H4001 with &H01, &H4002 with &H39.
```

The 'loader' will be used at the start of all the BASIC demonstration programs below.

3. Executing machine code programs.

There are two BASIC commands to start a machine code program on the DRAGON; EXEC and then user defined functions USR1 - USR9 . In all the programs below, the USR command will be used as this allows values to be returned to BASIC. The easiest way to return values to BASIC is to load the D register with a two's complement 16 bit integer and call the ROM routine GIVEABF immediately before the RTS instruction. This routine starts at &H8C37 and causes the number in the D register to be returned to BASIC.

## THE PROGRAMS

The programs that follow will be described in assembly format and a DATA list for use with the 'loader' above.

The instructions can loosely be grouped together under the following headings:

- Complete Byte and Register Handlers
- Arithmetic
- Logical
- Comparisons

Branches  
 Rotaters  
 Stack Handlers  
 Interrupt Handlers

Each program demonstrates one or more instructions from one of the groups.

### a) The No Operation Instruction

This is the simplest of all machine instructions as it does nothing. One or more NOP's followed by a 'return' constitutes a complete (though not very useful) program.

Program 1:

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	12	NOP	A single No Operation
4001	39	RTS	'return to BASIC'

Lines 10 - 70, the 'loader' as given above.

80 A = USR01(X)

90 PRINT "THE NOP MACHINE CODE PROGRAM HAS BEEN

EXECUTED"

500 DATA 12, 39, \*\*

### b) The complete Byte and Register Handlers

Load's - The following group of programs (1 - 4) demonstrates the range of the Load instructions.

Program 2:

This program demonstrates the Load into an 8-bit register in the immediate mode, i.e. the data to load is in the program.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #\$00	Make the A register zero
4002	C6 xx	LDB #\$xx	The user will enter values for xx
4004	BD 8C37	JSR \$8C37	Set up the D register to be returned
4006	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above

80 INPUT "ENTER A VALUE FOR THE B REGISTER (0 -- 255)";A

90 POKE &H4003, A

100 PRINT "THE D REGISTER NOW HOLDS" USR01(X)

110 GOTO 80

500 DATA 86, 00, C6, 00

510 DATA BD, 8C, 37, 39, \*\*

Program 3:

This program demonstrates the Load instruction into a 16-bit



register in the extended mode, i.e. the data to load is in memory. Note that direct addressing is very similar.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	FC 4100	LDD #\$4100	Make the D register the same as memory location \$4100 and \$4101
4003	BD 8C37	JSR \$8C37	Set up the D register to be returned
4006	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above

```

80 INPUT "ENTER A VALUE FOR MEMORY LOCATIONS $4100 AND
$4441 (0 - 6533)"; A
90 POKE &H4100, INT(A/256)
100 POKE &H4101, A-256*INT(A/256)
110 PRINT "THE D REGISTER NOW HOLDS" USR01(X)
120 GOTO 80
500 DATA FC, 41, 00
510 DATA BD, 8C, 37, 39, **

```

#### Program 4:

This program demonstrates Load into an 8-bit register using the indexing mode, i.e. the data is to be found at the memory location which is in one of the index registers:

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	8E 4100	LDX #\$4100	Make the X register 'point' to where the data is going.
4003	86 00	LDA #\$00	Make the A register 0
4005	E6 84	LDB ,X	Make the B register the same as the memory location which is in the X register
4007	BD 8C37	JSR \$8C37	Set up the D register to be returned
400A	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above

```

80 INPUT "ENTER A VALUE FOR $4100 (0 - 255)
90 POKE &H4100, A
100 PRINT "THE D REGISTER NOW HOLDS " USR01(X)
110 GOTO 80
500 DATA 8E, 41, 00, 86, 00, E6, 84
510 DATA BD, 8C, 37, 39, **

```

Program 5:

This program demonstrates the EXchanGe instruction by loading the X register (extended) then EXchanGing the X and D registers.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	BE 4100	LDX \$4100	Load the X register with what is in memory locations \$4100 and \$4101
4003	1E 01	EXG D, X	Exchange the D and X registers - D now equals what X used to and X now equals what D used to
4005	BD 8C37	JSR \$8C37	Set up the D register to be returned
4008	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above

```
80 INPUT "ENTER A NUMBER FOR THE X REGISTER (0 -
65535)"; A
90 POKE &H4100, INT(A/256)
100 POKE &H4101, A-256*INT(A/256)
110 PRINT "THE D REGISTER NOW HOLDS" USR01(X)
120 GOTO 80
500 DATA BE, 41, 00, 1E, 01
510 DATA BD, 8C, 37, 39, **
```

Program 6:

This program demonstrates the STore operations. Note that only the extended mode is demonstrated but direct and indexing modes can also be used.

The program loads an 8-bit register with a value then stores this in memory location \$4100.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 xx	LDA #\$xx	Load the A register with data entered by the user
4002	B7 4100	STA \$4100	Put the contents of the A register into memory location \$4100
4445	39	RTS	Return to BASIC

```

Lines 10 - 70, the 'loader' as given above
  80 INPUT "ENTER A NUMBER TO BE STORED (0 - 255)"; A
  90 POKE &H4001, A
 100 A = USR01(X)
 110 PRINT "THE MEMORY LOCATION $4100 NOW EQUALS"
PEEK(&H4100)
 120 GOTO 80
 500 DATA 86, 00, B7, 41, 00
 510 DATA 39, **

```

#### Program 7:

This program demonstrates the last of the load/store type instructions and that is the Load Effective Address. This will also demonstrate the indexing mode, as LEA can only use the indexing mode. The user will be asked to supply the information for the effective address port. This program can be used to check that you are using the right post-byte for your indexed addressing mode for other instructions. Note that this program only handles three byte indexed instruction (i.e. two-byte effective address, eg. 8-bit offset from PC)

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	30 0000	LEAX xxxx	Load the X register with the user-supplied effective address
4003	1F 10	TFR X, D	Put the contents of the X register into the D register (this is necessary as only the D register can be returned to BASIC)
4005	BD 8C37	JSR \$8C37	Set up the D register to be returned
4008	39	RTS	Return to BASIC

```

Lines 10 - 70, the 'loader' as given above
  80 INPUT "ENTER THE FIRST BYTE OF THE EFFECTIVE ADDRESS
(IN HEX 0 - FF); A$
  90 INPUT "ENTER THE SECOND BYTE OF THE EFFECTIVE ADDRESS
(IN HEX 0 - FF)"; B$
 100 POKE &H4001, VAL("&H" + A$)
 110 POKE &H4002, VAL("&H" + B$)
 120 PRINT "THE EFFECTIVE ADDRESS WAS " USR01(X)
 130 GOTO 80
 500 DATA 30, 00, 00, 1F, 10
 510 DATA BD, 8C, 37, 39, **

```

## c) The Arithmetic Instructions

These instructions are the ones which do the 'sums' for your programs. They are all either inherent address (i.e. need no data) or use one or more registers and some data.

### Program 8:

This program demonstrates the Add the B register to the X register instruction by asking for numbers to go into both B and X registers, performing the addition and returning the number through the D register. ABX is an inherent instruction as it only uses the B and X registers.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 00	LDB # $\$xx$	Load B register with user-supplied number
4002	8E 0000	LDX # $\$xxxx$	Load X register with user-supplied number
4005	3A	ABX	Add B to X leaving the answer in X register
4006	1F 10	TFR X,D	Copy the contents of X to D register
4008	BD 8C37	JSR $\$8C37$	Set up the D register to be returned
400B	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A VALUE FOR THE B REGISTER (0 - 255)";A
90 INPUT "ENTER A VALUE FOR THE X REGISTER (0 -
65535)"; B
100 POKE &H4001, A
110 POKE &H4003, INT(B/256) : POKE &H4004,
B-256*INT(B/256)
120 PRINT "THE ANSWER IS "USR01(X)
130 GOTO 80
500 DATA C6, 00, 8E, 00, 00, 3A, 1F, 10
510 DATA BD, 8C, 37, 39, **
```

### Program 9:

This program demonstrates the Add with Carry instruction by setting the carry bit and adding two user-supplied numbers and the carry. The ADC instruction can use either the A or B registers and a memory location with any of the addressing modes (i.e. direct, extended, immediate, indexed).

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA # $\$00$	Make A zero
4002	C6 00	LDB # $\$xx$	Load B with user-supplied number
4004	1A 01	ORCC # $\$01$	Set the carry bit
4006	C9 00	ADCB # $\$xx$	Add B and Carry and user-supplied number
4008	BD 8C37	JSR $\$8C37$	Set up the D register to be returned
400B	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A VALUE FOR THE B REGISTER (0 - 255)";
A
90 INPUT "ENTER A VALUE TO BE ADDED TO THE B REGISTER
(0 - 255)"; B
100 POKE &H4003, A
110 POKE &H4007, B
120 XX = USR01(X)
130 PRINT "THE B REGISTER NOW HOLDS ";
XX-256*INT(XX/256)
140 GOTO 80
500 DATA 86, 00, C6, 00, 1A, 01, C9, 00
510 DATA BD, 8C, 37, 39, **
```

Try this program after changing the ORCC #01 (1A 01) to ANDCC #0FE (1C FE) which resets the carry bit.

Program 10:

This program demonstrates the ADD instruction which is similar to the ADC instruction except that the Carry bit is not referenced. The ADD instruction can use any of A, B or D registers and a memory location in any of the addressing modes.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #00	Make A zero
4002	C6 00	LDB #yy	Load B with user-supplied number
4004	CB 00	ADD B #00	Add user-supplied number to B
4006	BD 8C37	JSR \$0C37	Set up the D register to be returned
4009	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A VALUE FOR THE B REGISTER (0 - 255)";
A
90 INPUT "ENTER A VALUE TO BE ADDED TO THE B REGISTER
(0 - 255)"; B
100 POKE &H4003, A
110 POKE &H4005, B
120 XX = USR01(X)
130 PRINT "THE B REGISTER NOW HOLDS " XX-256*INT(XX/256)
140 GOTO 80
500 DATA 86, 00, C6, 00, CB, 00
510 DATA BD, 8C, 37, 39, **
```

Program 11:

This program demonstrates the Decimal Addition Adjust instruction by doing a decimal adjust on the result of a user-supplied number added to 2 (i.e. no change if user number < 8). DAA is an inherent instruction as it only uses the A register.

ADDRESS	OPCODE	MNEMONIC	COMMENT
---------	--------	----------	---------

400J	86 00	LDA #xx	Load A with user-supplied number
4002	C6 00	LDB #00	Make B zero
4004	8B 02	ADDA #C2	Add 2 to the A register
4006	19	DAA	Decimal Adjust the A register
4007	BD 8C37	JSR \$8C37	Set up the D register to be returned
400A	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```

80 INPUT "ENTER THE NUMBER TO BE ADJUSTED (0 - 255)"; A
90 POKE &H4001, A
100 XX = USR01(X)
110 PRINT "THE NUMBER AFTER ADJUSTMENT IS ", INT(XX/256)
120 GOTO 80
500 DATA 86, 00, C6, 00, 10
510 DATA BD, 8C, 37, 39, **

```

#### Program 12:

This program demonstrates the DECRement command which subtracts one from either the A or B register or a memory location. DEC is inherent for registers A and B or can use any of the addressing modes (except for immediate) on a memory location.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #00	Make A zero
4002	C6 00	LDB #xx	Load B with user-supplied number
4004	5A	DECB	Decrement the B register
4005	BD 8C37	JSR \$8C37	Set up the D register to be returned
4008	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```

80 INPUT "ENTER A VALUE TO BE DECREMENTED"; A
90 POKE &H4003, A
100 XX = USR01(X)
110 PRINT "THE B REGISTER NOW HOLDS " XX-256*INT(XX/256)
120 GOTO 80
500 86, 00, C6, 00, 5A
510 DATA BD, 8C, 37, 39, **

```

#### Program 13:

This program demonstrates the INCRement instruction which is exactly the opposite of DEC as it adds one to A, B or a memory location.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #00	Make A zero
4002	C6 00	LDB #xx	Load B with user-supplied number
4004	5C	INCB	Increment B
4005	BD 8C37	JSR \$8C37	Set up the D register to be returned

```
4008 39 RTS Return to BASIC
```

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT 'ENTER A VALUE TO BE INCREMENTED (0 - 255)"; A
90 POKE &H4003, A
```

```
100 PRINT "THE B REGISTER NOW CONTAINS "
```

```
XX-256*INT(XX/256)
```

```
110 GOTO 80
```

```
500 DATA 86, 00, C6, 00, 5C
```

```
510 DATA BD, 8C, 37, 39, **
```

#### Program 14:

This program demonstrates the MULTIPLY instruction which takes two numbers, in the A and B registers, and multiplies them together returning the result in the D register. The MUL instruction is inherent as it always uses these 3 registers. Note that D is really A and B register used together.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #xx	Load A with first number
4002	C6 00	LDB #yy	Load B with second number
4004	3D	MUL	Multiply A and B together
4005	BD 8C37	JSR \$8C37	Set up the D register to be returned
4008	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A VALUE FOR THE A REGISTER (0 - 255)";
```

A

```
90 INPUT "ENTER A VALUE FOR THE B REGISTER (0 - 255)";
```

B

```
100 POKE &H4001, A
```

```
110 POKE &H4003, B
```

```
120 PRINT "THE B REGISTER NOW HOLDS " USR01(X)
```

```
130 GOTO 80
```

```
500 DATA 86, 00, C6, 00, 3D
```

```
510 DATA BD, 8C, 37, 39, **
```

#### Program 15:

This program demonstrates the NEGATE instruction which takes a 2's complement number and negates it, i.e. toggles all the bits and adds one. NEG can be used inherently on the A and B registers or any of the other addressing modes, except immediate, for a memory location.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #0C	Make A zero
4002	C6 00	LDB #xx	Load B with user-supplied number
4004	50	NEGB	Negate register B
4005	BD 8C37	JSR \$8C37	Set up the D register to be returned
4008	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A VALUE TO BE NEGATED (0 - 255)"; A
```

```

90 PCKE &H4003, A
100 XX = USR01(X)
110 PRINT "THE B REGISTER NOW HOLDS " XX-256*INT(XX/256)
120 GOTO 80
500 DATA 86, 00, C6, 00, 50
510 DATA BD, 8C, 37, 39, **

```

Program 16:

This program demonstrates the SuBtract with Carry instruction by loading the B register with the number to subtract from, setting the carry bit, then performing the subtraction. This demonstration uses the immediate mode and the B register. Either the A or B register can be used with any but the inherent addressing modes. Try the program without the carry bit set by changing the ORCC #01 (1A 01) to and ANDCC #0FE (1C FE).

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #00	Make A zero
4002	C6 00	LDB #xx	Load B with number to be subtracted from
4004	1A 01	ORCC #01	Set the carry bit
4006	C2 00	SBCB #yy	Subtract a user-supplied number from the B register
4008	BD 8C37	JSR \$8C37	Set up the D register to be returned
400B	39	RTS	Return to BASIC

Lines 10 - 70, the 'loader' as given above.

```

80 INPUT "ENTER A NUMBER FOR THE B REGISTER (0 - 255)";
A
90 INPUT "ENTER A NUMBER TO SUBTRACT FROM B"; B
100 POKE &H4003, A
110 POKE &H4007, B
120 XX = USR01(X)
130 PRINT "THE B REGISTER NOW CONTAINS "
XX-256*INT(XX/256)
140 GOTO 80
500 DATA 86, 00, C6, 00, 1A, 01, C2, 00
510 DATA BD, 8C, 37, 39, **

```

Note that the SUB instruction is very similar except that it does not take the carry bit into account and it can be used with a 16-bit number using the D register.

Program 17:

This program demonstrates the Sign EXTend instruction. SEX extends the sign bit of the B register into the A register therefore it is an inherent instruction and does not need any data.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #00	Make A zero
4002	C6 00	LDB #xx	Load B with user-supplied number
4004	1D	SEX	Sign extend B into A



```
4005    BD 8C37 JSR $8C37 Set up the D register to
                                be returned
4008    39      RTS      Return to BASIC
```

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER THE NUMBER TO BE EXTENDED (0 - 255)"; A
90 POKE &H4003, A
100 XX = USR01(X)
110 PRINT "THE A REGISTER NOW CONTAINS " INT(XX/256), "
AND THE B REGISTER NOW CONTAINS " XX-256*INT(XX/256)
120 GOTO 80
500 DATA 86, 00, C6, 00, 1D
510 DATA BD, 8C, 37, 39, **
```

## d) The Logical Instructions

This series of programs demonstrates the logical instructions by asking for two numbers which you enter, in decimal as before. These numbers are then displayed along with their hexadecimal and binary equivalents followed by the result in the same format.

Program 18:

The AND instruction can be used with the A, B or CC registers and any addressing mode, except inherent, for the second operand.

NOTE: The BIT test instruction has the same effect on the CC register but it does not effect either operand.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 00	LDB #\$xx	Load the first number into the B register.
4002	C4 00	ANDB #\$yy	AND the B register with the second number.
4004	BD 8C37	JSR \$8C37	Set up the D register to be returned.
4007	39	RTS	Return to BASIC.

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A NUMBER FOR THE B REGISTER (0 - 255)";
A
90 INPUT "ENTER A NUMBER TO BE ANDED WITH THE B
REGISTER (0 - 255)"; B
100 POKE &H4001, A : POKE &H4003, B
110 F$ = " ! ### % % %"
120 PRINT "    DEC  HEX  BINARY"
130 NU = A : GOSUB 300
140 PRINT USING F$; "A", A, HLX$(A), NU$
150 PRINT "AND"
160 NU = B : GOSUB 300
170 PRINT USING F$; "B", B, HEX$(B), NU$
180 PRINT
190 XX = USR01(X)
200 NU = XX : GOSUB 300
210 PRINT USING F$; "=", XX, HEX$(XX), NU$
220 GOTO 80
300 REM MAKE NU$ BE THE BINARY OF NU
310 NU$ = ""
320 FOR I = 0 TO 7
330 NU$ = STR$(SGN(NU AND 2^I)9 + NU$)
340 NEXT I
350 RETURN
500 DATA C6, 00
510 DATA C4, 00
520 DATA BD, 8C, 37, 39, **
```

Note that this program will be used for other instructions in this group by changing lines 150 and 510.

Program 19:

The Exclusive OR instruction can use the A or B register and any addressing mode for the other operand. The same program as for AND can be used to demonstrate this by changing line 150 to read PRINT "EOR" and line 510 to read DATA C8, 00.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 00	LDB #\$xx	Load the first number into the B register.
4002	C8 00	EOR #\$yy	Exclusive OR B with the second number.
4004	BD 8C37	JSR \$8C37	Set up the D register to be returned.
4007	39	RTS	Return to BASIC.

Lines 10 - 140 as per Program 18

150 PRINT "EOR"

Lines 160 - 500 as per Program 18

510 DATA C8, 00

Line 520 as per Program 18

Program 20:

The OR instruction has exactly the same characteristics as the AND instruction except that it performs a logical OR on the bits instead of an AND. The same program can be used, again changing lines 150 and 510.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 00	LDB #\$xx	Load the first number into the B register.
4002	CA 00	ORB #\$yy	OR the B register with the second number.
4004	BD 8C37	JSR \$8C37	Set up the D register to be returned.
4007	39	RTS	Return to BASIC.

Lines 10 - 140 as per Program 18

150 PRINT "OR"

Lines 160 - 500 as per Program 18

510 DATA CA, 00

Line 520 as per Program 18

Program 21:

The Complement instruction simply "toggles" (i.e. changes all the bits previously 1 to 0 and vice-versa) all the bits in either the A or B registers or a memory location using any of the addressing modes.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 0C	LDB #\$xx	Load the first number into the B register.
4002	53	COMP	Complement the B register
4003	BD 8C37	JSR \$8C37	Set up the D register to be returned.

4006 39 RTS Return to BASIC.

Lines 10 - 70, the 'loader as given above

80 INPUT "ENTER THE NUMBER TO BE COMPLEMENTED (0 - 255)"; A

```
90 POKE &H4001, A
100 REM
110 F$ = " ! ### %% % %"
120 PRINT " DEC HEX BINARY"
130 NU = A : GOSUB 300
140 PRINT USING F$; "B", A, HEX$(A), NU$
150 XX = USR01(X)
160 NU = XX : GOSUB 300
170 PRINT "COM"
180 PRINT USING F$; "=", XX, HEX(XX), NU$
190 GOTO 80
300 REM MAKE NU$ BE THE BINARY OF NU
310 NU$ = ""
320 FOR I = 0 TO 7
330 NU$ = STR$(SGN(NU AND 2^I)9 + NU$)
340 NEXT I
350 RETURN
500 DATA C6, 00, 53
510 DATA BD, 8C, 37, 39, **
```

## e) Comparisons

Program 22:

The COMPARE instruction can inherently use any register for the first operand and any addressing method to get the second. The demonstration program displays both the B register (to show it hasn't been changed) and the CC register to show it has been changed. Note that neither of the numbers (register or memory location) is affected in any way, only the CC register is changed.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 00	LDB #\$\$xx	Load the B register with the number to be compared
4002	C1 00	COMPB #\$\$yy	Compare B register with user-supplied number
4004	1F A8	TFR A, CC	Load A register with CC register to be returned
4006	BD 8C37	JSR \$8C37	Set up the D register to be returned.
4009	39	RTS	Return to BASIC.

Lines 10 - 70, the 'loader' as given above

80 INPUT "ENTER A NUMBER FOR THE B REGISTER (0 - 255)";

A

90 INPUT "ENTER THE NUMBER TO BE COMPARED WITH THE B REGISTER"; B

```
100 POKE &H4001, A : POKE &H4003, B
110 XX = USR01(X)
```

```

120 PRINT "AFTER THE COMPARISON:"
130 PRINT "THE B REGISTER HOLDS " XX-256*INT(XX/256)
140 NU = INTT(XX/256) : GOSUB 300
150 PRINT "AND THE CC REGISTER HOLDS" NU$
160 PRINT TAB(29), " E F H I N Z V C"
170 GOTJ 80
Lines 300 -360 as per Program 18
500 DATA C6, 00
510 DATA C1, 00
520 DATA 1F, A8
530 DATA BD, 8C, 37, 39, **

```

Program 23:

The Test instruction sets the CC register according to the operand. The operand can be either the A or B register or a memory location using any of the addressing modes except immediate. As it is similar to the COMPare (i.e. it only changes the CC register) the program used for COM will be modified to suit TST.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	C6 00	LDB #\$xx	Load the B register with the number to be tested
4002	5D	TESB	Test the B register
4004	1F A8	TFR A, CC	Load A register with CC register to be returned
4006	BD 8C37	JSR \$8C37	Set up the D register to be returned.
4009	39	RTS	Return to BASIC.

Lines 1C - 70, the 'loader' as given above  
80 INPUT "ENTER A NUMBER FOR THE B REGISTER (0 - 255)";

A

```

100 POKE &H4001, A
110 XX = USR01(X)
120 PRINT "AFTER THE TEST:"
Lines 130 - 500 as per Program 23
510 DATA 5D
520 DATA 1F, A8
530 DATA BD, 8C, 37, 39, **

```

NOTE: make sure to delete line 90!

## f) The Branch and Jump Instructions

The 22 instructions in this group allow the user to make jumps from one part of a machine code program to another. These jumps are broken up into two types; they are: branches and jumps. Branches make jumps relative to the current program counter (PC), i.e. the new PC value is the old PC value plus the operand of the instruction. Every branch instruction can have either an 8-bit or 16-bit operand and usually branches on conditions in the CC register. The jumps use absolute addressing, i.e. the new PC value is the operand of the instruction and this is always a 16-bit number.

There are two special instructions, one branch and one jump, which go to subroutines and these are covered in Program 24.

Program 23 is designed to be used for all the other jumps/branches.

Program 23:

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA #\$00	Set A to zero
4002	C6 00	LDB #\$xx	Load two values to be
4004	C1 00	CMPB #\$yy	compared so as to set
			the flags
4006	24 07	BCC NEXT	Branch forward to NEXT
4008	12	NOP	For later use
4009	BD 8C37	JSR \$8C37	Set up the D register to
			be returned
400C	39	RTS	Return if no jump made
400D	12	NOP	Two unused
400E	12	NOP	locations
400F	4C NEXT	INC A	Increment A if jump made
4010	BD 8C37	JSR \$8C37	Set up the D register to
			be returned
4013	39	RTS	Return to BASIC

The above program will return a 1 in the A register if the jump was made, otherwise it will return 0 in the A register.

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER TWO VALUES TO BE COMPARED (0 - 255)";
A, B
90 POKE &H4003, A : POKE &H4005, B
100 XX = USR01(X)
110 IF INT(XX/256) THEN PRINT "JUMP MADE" ELSE PRINT
"JUMP NOT MADE"
120 GOTO 80
500 DATA 86, 00, C6, 00, C1, 00
510 DATA 24, 07, 12
520 DATA BD, 8C, 37, 39, 12, 12, 4C
530 DATA BD, 8C, 37, 39, **
```

To use the above program to test other branch/jump instructions the only line that needs to be changed is line

number 510. The changes are as follows:

BCS	510 DATA 25, 07, 12
BEQ	510 DATA 27, 07, 12
BGE	510 DATA 2C, 07, 12
BGT	510 DATA 2E, 07, 12
BHI	510 DATA 22, 07, 12
BHS	510 DATA 24, 07, 12
BLE	510 DATA 2F, 07, 12
BLO	510 DATA 25, 07, 12
BLS	510 DATA 23, 07, 12
BLT	510 DATA 2D, 07, 12
BMI	510 DATA 2B, 07, 12
BNE	510 DATA 26, 07, 12
EPC	510 DATA 2A, 07, 12
BRA	510 DATA 20, 07, 12
BRN	510 DATA 21, 07, 12
BVC	510 DATA 28, 07, 12
BVS	510 DATA 29, 07, 12
JMP	510 DATA 4E, 70, 0F

#### Program 24

This program demonstrates the last instructions in this group, namely the BSR, JSR and RTS instructions. As BSR and JSR are identical except in their addressing mode, only the BSR will be demonstrated. The program jumps to a subroutine which increments the A register then returns to the main program.

ADDRESS	OPCODE	MNEMONIC	COMMENT
4000	86 00	LDA # $\$xx$	Load the A register with a user-supplied number
4002	8D 07	BSR INCR	Branch to the subroutine called INCR
4004	12	NOP	Location unused
4005	BD 8C37	JSR $\$8C37$	Set up the D register to be returned
4008	39	RTS	Return to BASIC
4009	12	NOP	Two unused
400A	12	NOP	locations
4004	4C INCR	INC A	Increment A if jump made
4013	39	RTS	Return to main program

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A NUMBER TO BE INCREMENTED (0 - 255)";
```

A, B

```
90 POKE &H4001, A
100 XX = USR01(X)
110 PRINT "THE A REGISTER NOW CONTAINS " INT(XX/256)
120 GOTO 80
500 DATA 86, 00, 8D, 07, 12
510 DATA BD, 8C, 37, 39, 12, 12
520 DATA BD, 8C, 37, 39, **
```

## g) The Rotate Instructions

The instructions in this group are very similar as they rotate or shift all the bits in either the A or B register or a memory location using any of the addressing modes except immediate. For this reason only one program will be written and a list of changes to be made for the other instructions will follow. The program will ask the user for a number, and whether to set or reset the carry bit. It will then display the number in decimal, hexadecimal and binary and the carry bit, then rotate or shift the number and display it, again in the same format.

Program 25:

ADDRESS	OFCODE	MNEMONIC	COMMENT
4000	1A 01	ORCC	Set the carry bit
4002	78 4100	ASL \$4100	Rotate the user-supplied number
4005	1F A9	TFR CC,B	Transfer the CC register be returned
4007	39	RTS	Return to BASIC

Lines 10 - 79, the 'loader' as given above.

```
80 INPUT "ENTER THE NUMBER TO ROTATE (0 - 255)"; A
90 INPUT "CARRY BIT SET OR RESET (S - R)"; B$
100 POKE &H4003, A
110 IF B$ = "R" THEN POKE &H4000, &H1C : POKE &H4001,
```

&HF

```
120 IF B$ ( ) "S" THEN 90
130 POKE &H4000, &H1A : POKE &H4001, &H01
140 PRINT "ASL/LSL"
150 F$ = " ### %%"
160 PRINT "BEFORE"; : GOSUB 200
170 XX = USR01(X)
180 PRINT "AFTER"; : GOSUB 200
190 GOTO 80
200 NU$ = "" : NU = PEEK(&H4001)
210 FOR I = C TO 7
220 NU$ = STR$(SGN(NU AND I+2))
230 NEXT I
240 PRINT USING F$; NU; HEX$(NU); NU$
250 IF XX AND 1 THEN PRINT "CARRY BIT SET" ELSE PRINT
"CARRY BIT RESET"
260 RETURN
500 DATA 1A, 01
510 DATA 78
DATA 71, 00, 1F, A9, 39
```

For the other rotate instructions make the following changes to lines 140 and 510.

```
140 PRINT "ASR" 510 DATA 77
140 PRINT "LSR" 510 DATA 74
140 PRINT "ROL" 510 DATA 79
140 PRINT "ROR" 510 DATA 76
```



## h) The Stack Handling Instructions

The program below will only be a simple demonstration of what these instructions do. It will prompt the user for a number which it will put into the X index register. The X register will be pushed onto the user stack then the D register will be pulled from the user stack and the number displayed. The reader is encouraged to experiment further with the PSH and PUL instructions but be sure that there is the same number of PUL's as there are PSH's.

Program 26:

ADDRESS	OF CODE	MNEMONIC	COMMENT
4000	8E 0000	LDX # $\$xxxx$	
4003	CE 4100	LDU # $\$4100$	
4006	36 10	PSHU X	
400E	37 06	PULU D	
400A	BD 8C37	$\$8C37$	
400D	39	RTS	

Lines 10 - 70, the 'loader' as given above.

```
80 INPUT "ENTER A NUMBER TO BE PUSHED ONTO THE U STACK
(O - 255)"; A
90 POKE &H4001, A-256*INT(A/256)
100 POKE &H4002, INT(A/256)
110 PRINT "THE NUMBER ON TOP OF THE U STACK IS "USR01(X)
120 GOTO 80
500 DATA 8E, 00, 00, CE, 41, 00
510 DATA 36, 10, 37, 06
520 DATA BD, 8C, 37, 39, **
```

## i) The Interrupt Instructions

These instructions are not demonstrated but the reader can write his own routine which uses these instructions.

## CHAPTER 15

# Programming Your Dragon

## Planning Your Machine Language Programs

Machine language programming is extremely flexible in that it allows you to do just about anything at all.

Since all higher level languages ultimately have to come down to machine language, it follows that anything you can program in FORTRAN or COBOL or any other language can be done in machine language. With the additional advantage that the machine language program will be the faster one.

This total flexibility can however also be a trap to the unwary programmer. With so much freedom, it is possible to do anything. Unlike the DRAGON's operating system (BASIC), for example, there are no checks on whether a statement is a legal one.

Since all numbers you can enter will be an instruction of one kind or another, the 6809 will process anything.

But beyond the problems of checking whether the syntax is legal, machine language programming has no constraints on your logic - you can perform functions, jumps, etc. which would be totally illegal in any higher level language.

It is therefore of the utmost importance to discipline yourself in the design of your program. We cannot recommend too highly the concept of the 'top-down' approach in design, especially in the design of machine language programs.

The 'top-down' approach forces you to break the problem into smaller units, and enables you to check the logic of your design without doing any coding for a long time.

Suppose you wanted to write a lunar lander program:  
The very first step might look something like this:

```
INSTR      Display instructions
           jump back to INSTR till ENTER pressed

DRAW      Draw landscape, start lander at top

LAND      Move lander
           If fuel finished go to CRASH
           Jump back to INSTR if not ground

GROUND    Print congratulations
           Jump back to INSTR for next go

CRASH     Print Commiserations on bad landing
           Jump back to INSTR for next go
```

Notice how this 'program' is written totally in English. At this stage, no decision has been made whether the program is to be written in BASIC or machine language. Nor is it necessary to make that decision - the concept of the lunar lander program is not dependent on the coding.

Now comes the part of logic testing. You play the part of the computer and see if all the possibilities you wish to see in the program are covered.

Are there any jumps to code that you meant to write but forgot? Are some routines redundant? Should some of them be put into subroutines?

Let us look at the 'program' again - we forgot to allow for a finish to the program!

The above logic might be fine for some applications, such as an arcade machine, but in your programs you may decide you would like to stop the program.

We now change the last part of the program as follows:

```
GROUND      Print congratulations
              Jump to finish
CRASH       Print commiserations on bad landing
FINISH      Ask player if finished
              If not, jump to INSTR
              If yes, STOP
```

Note that we have used labels to describe certain lines in the program. These are very valuable, the more so if you choose short labels which are descriptive in meaning.

Once this level is finished, you move down a level to do the same thing to one of the lines or modules above. This is why this approach is called 'top-down'.

For example we can expand the 'finish' module above:

```
FINISH      Clear screen
              Print "Would you like to stop now?"
              Scan keyboard for input
              If input = yes then stop
              Jump to INSTR
```

The benefit of the top-down approach is that you can test and run a particular module on its own, so that it is ready for the final program.

Lets us go down another level again, and look at the "Clear screen" line in more detail.

At this stage we do have to decide on what language we will write the program in, and let us choose machine language on the DRAGON.

If you were writing in BASIC, all you would have to say is:  
900 CLS  
but in machine language that simple sentence, 'clear screen' can be deceptive.

We might therefore do something like this:

```
CLEAR      Find the start of screen
           Fill the next 6144 positions with blanks
```

We still haven't done any coding, but obviously the approach is based on machine language. Now is the time to look more closely at exactly what this clear screen routine is meant to do and what it actually does do, by again pretending to be the computer.

It seems that it does everything that it is meant to so we continue on to the next stage.

The next level down is the one where you must finally do the coding, so let us look at filling the screen with blanks:

```
CLEAR      LDX  $$0600      Start of screen
           LDD  $$0000      D = blanks
LOOP       STD  ,X++        Blank current position
           and move to the next
           position
           CMPX $$1E>0     Test, end of screen?
           BNE  LOOP       No, go back again
```

You should be able to deal with programs of this length quite easily and in this way build up very complex programs. By the way, you no doubt understand now, why machine language programs tend to be so long and why people invented higher level languages!

## Entering and Running Machine Language Programs

There are two ways to enter information directly into memory: POKing and CLOADing. These two commands allow you to actually set memory locations to certain values.

If you are not going to be writing a multitude of machine language programs the easiest way to store, enter and run your programs, while they are being developed, is to have them as DATA in a BASIC program. This makes editing them quite simple especially if you use a convention such as only having one instruction/operand per DATA statement. When your machine language program is finished (running correctly), immediately after RUNNING the BASIC program, CSAVEM your machine language program for future use. Remember, whenever you are mixing BASIC programs with machine code programs, to set the upper limit that BASIC can use with the CLEAR statement, otherwise the programs can destroy each other.

On the other hand if you are going to be doing lot of machine language programming then you need a monitor program, or better still a full assembler/editor. If you look around you will find a few assembler/editors on the market but if you want to get out of it cheaply, I have included a monitor program with which you can enter, modify, and execute a machine language program as well as convert numbers from hexadecimal to decimal and vice-versa.

The commands for the monitor, their formats and a description of each is given below followed by the program listing.

M - M(address) - Memory examine and change.

- when this command is used, the address, the contents of the address and a hyphen are displayed. The two arrow keys on the left, display the next higher and lower address respectively. If at any time you want to change the contents of the address being displayed hit C and enter the value (in hex). To return to command level key ENTER directly after a hyphen is displayed.

D - D(address) - Display memory

- This will display the memory in groups of four bytes followed on the same line by the four characters which represent the memory's contents. After one screen full of memory is displayed, the listing stops and waits for you to select an option - a space will give you another screen full of information - any other key will return to the command mode.

F - F(b address) (e address) (string) - Find a string in memory.

- This searches memory beginning at (b address) through to (e address) for the character string, (string). All the addresses which point to the start of (string), if there are any, are displayed.

C - C D (number)  
- C H (number)  
- This converts the number (number) to either hex or decimal depending on whether H(decimal to hex) or D(hex to decimal) is specified.

J - J(address) - Jump to machine language program  
- This causes the machine language program starting at (address) to be executed.

E - E - Exit to BASIC.

NOTE: All addresses must be four hex digits.  
Formats must be exactly as shown, i.e. no spaces between the command letter and the first address (except for C) and 1 space between all other parameters.

The next section contains sample machine language programs for you to use, either as part of your own programs or an example of what can be done in machine code. These programs can be entered in any of the methods mentioned above. If you are using the above monitor program or similar, then you need to enter the actual hex bytes. If you have an assembler then you only need to enter the mnemonics (instructions) and operands (you might like to enter the comments for future reference as well).

The assembler listings supplied have the following format:

COLUMN 1: Program line number. Produced by the assembler. You do not have to key this in.

COLUMN 2: Memory address. Produced by the assembler. You have to take note of this if you are not using an assembler so you know where to enter the hex bytes. When using the above monitor you only need to enter the first address and the program automatically moves onto the next address, it is a good idea to occasionally check that you are putting the right numbers in the right addresses.

COLUMN 3: Instruction & operand. Produced by the assembler. This can be up to four, two digit hex numbers. The first number goes in the address immediately to it's left, the next number goes in that address plus one, etc.

THE REST: Mnemonic, operand and comments. This is what the assembler uses to produce the above columns. If you are using an assembler then this is all you need. If you are not using an assembler then you need not enter this at all.

#### EXAMPLE OF HOW TO ENTER PROGRAM LINES

This is a step by step example of how to enter a line of a program from the assembler listing. The line to be used as an example is line number 45 in the PIA program. The line is:  
0045 408F A623 DDRB LDA 3,Y Set DDRB select bit to 0

When using an assembler:

If your assembler requires you to enter in your own line numbers, then you need to type 0045, otherwise ignore it. Next type 'DDRB LDA 3,Y'. This is a label (DDRB), an instruction (LDA) and an operand (3,Y). Then if you wish to have comments in your program for future reference, type in 'Set DDRB select bit to 0'

When not using an assembler:

After loading and running a monitor or similar program and going into the mode that lets you enter hex bytes (M in the above monitor) you need to go through the assembler listing picking out the bytes (instruction and operands) that need to be entered and enter them. The hex bytes for the instruction and operand are located in the third column. Note that some lines have what is called pseudo operands which are instructions to the assembler not to the processor. Only enter the numbers when there is an address directly after the line number. Some times there are numbers in the address column that are not addresses and therefore the line should be ignored. All the programs in this book start at \$4000 so the addresses will start at 4000 and increase as the program goes on. The addresses increase continuously by the same number of bytes as there are on the line, this is usually 2 or 3 but can be more. As an example lines 0001, 0002 and 0003 (in program PIA) can be ignored as these tell the assembler where to start the addresses and the values of two constants (these can be ignored as you enter the bytes directly into the addresses and use actual numbers not labels). Line 0004 has an address in the second column so the two bytes (20, 7D) in the third column need to be entered at addresses \$4000 and \$4001 respectively. Lines 0005 and 0006 have addresses but no bytes so these addresses must be left blank (these are going to be used as variables), then line 0007 has an address and bytes to be entered. Let's pretend that we are now up to line 0045 (having just typed in line 0044), and it is time to check that you are doing the right thing. The monitor program should now display address 408F for you to change. If this is not the case then you have made a mistake in the number of bytes you have typed in and you need to look back and see where you went wrong. If the monitor is displaying the right address it still doesn't mean that you have typed in the byte correctly just the right number of them. It is probably best to have someone read out what you have entered while you read through the listing checking that they are the same. This check should be done periodically throughout your typing because if you do leave out or put in an extra byte the whole program needs to be retyped from the address at which you made the error.

Back to the example of line 0045. The monitor is displaying address 408F and waiting for you. You then have to change the contents of that address (the procedure will be different for different monitors - C for the one in the book) to A6 (the first byte in the third column). The monitor should then display address 4090 and wait for you again. The contents of



this address should be changed to 23 and then you are ready for the next line (0046).

Note that it is strongly recommended that you CSAVEM your program before trying to use it, otherwise it will have been a big waste of time typing it all in if it destroys itself.

So go to it and enjoy the wonderful world of machine language.

## Monitor Program

```
5 HD$="0123456789ABCDEF"
10 PRINT"COMMANDS :M, D, F, C, J, E"
20 INPUT CL$:CT$=LEFT$(CL$,1)
25 IF CT$=""THEN10
30 CT=INSTR("MDFCJE",CT$)
40 IF CT=0 THEN 10
50 ON CT GOSUB 100,300,400,500,600,700
60 GOTO10
100 BA=VAL("&H"+MID$(CL$,2,4))
110 IF BA<0 OR BA>&HFFFF THEN 900
120 AC=PEEK(BA)
130 PRINT HEX$(BA) "HEX$(AC)" - ";
140 MC$=INKEY$:IF MC$="" THEN140
150 MC=ASC(MC$)
160 IF MC=94 THEN BA=BA+1:PRINT:GOTO110
170 IF MC=10 THEN BA=BA-1:PRINT:GOTO110
185 IF MC=13 THEN PRINT:RETURN
190 IF MC$<>"C" THEN14C
200 INPUT MC$
220 NB=VAL("&H"+MC$):IF NB<0 OR NB>&HFF THEN PRINT:RETURN
230 POKEBA,NB
240 BA=BA+1:GOTO120
300 BA=VAL("&H"+MID$(CL$,2,4))
310 IF B<0 OR BA>&HFFFF THEN900
320 CNT=0
325 PRINTEX$(BA);
330 FORI=OTO3:PRINT"HEX$(PEEK(BA+I));:NEXTI
335 PRINT" ";
340 FORI=OTO3:PRINT"CHR$(PEEK(BA+I));:NEXTI:PRINT
345 BA=BA+4
350 IF CNT<14 THEN CNT=CNT+1:GOTO325
355 PRINT
360 CNT=0
370 MC$=INKEY$:IF MC$=""THEN 370
330 IF MC$="" THEN325ELSE RETURN
400 BA=VAL("&H"+MID$(CL$,2,4))
410 BE=VAL("&H"+MID$(CL$,7,4))
420 IF E<0 OR BA>&HFFFF OR BE<BA OR BE>&HFFFF THEN900
430 FS$=MID$(CL$,12):FL=LEN(FS$)
440 FORI=BA TO BE
450 IF PEEK(I)<>ASC(LEFT$(FS$,1))THEN490
460 FOR J = I TO I + FL-1 : IF PEEK ( J ) <> ASC ( MID$ (
FS$, J-I+1, 1 ) ) THEN 490
470 NEXTJ
480 PRINT HEX$(I)
490 NEXTI:RETURN
500 IF MID$(CL$,3,1)<>"D"THEN520
510 PRINT VAL("&H"+MID$(CL$,5)):RETURN
520 IF MID$(CL$,3,1)<>"H" THEN RETURN
530 PRINTEX$(VAL(MID$(CL$,5))):RETURN
600 BA=VAL("&H"+MID$(CL$,2))
610 IF BA<0 OR BA>&HFFFF THEN900
700 END
900 PRINT"ILLEGAL HEX ADDRESS"
910 END
```

## CHAPTER 16

# Sample Programs

### Introduction

The programs contained in this chapter have been written with two purposes in mind; to demonstrate machine language programming and to give examples of how to make the most out of your DRAGON using machine code.

Before giving you the actual programs there is a short description on each of the major chips, apart from the CPU. These chips include the PIA, VDG and the SAM for without these three chips the CPU would be totally senseless (without senses that is).

### The PIA (Peripheral Interface Adaptor)

The PIA is the main controller of the I/O (Input/Output) of the DRAGON. All communication with the outside world is channeled through the PIA or controlled by it in some way.

In the DRAGON there are two of these PIA chips and each PIA has two 'ports'. A port is like a gate where information (instead of cows) can pass through. Each port has three registers to control its behaviour - PDR, DDR, CR (Peripheral Data Register, Data Direction Register and the Control Register )

The PDR is the register which is the actual gate i.e. where the information passes through. The data in this register can be either input or output ( controlled by the DDR's ).

The DDR contains the direction of the PDR. This controls the direction that the data in the PDR is going. Each bit can be set irrespective of any others around it. This means that one PDR may have both input and output lines. The PDR's and their corresponding DDR's share the same address. To access the correct register a bit is set in the corresponding control register (see below).

The CR controls all sorts of things. It can control devices that are either on or off, it selects between PDR and DDR and also controls interrupts.

PIA 0 is located at address \$FF00 - \$FF03 and PIA 1 is located at address \$FF20 - \$FF22 but because they do not have a full address decoder, are repeated seven times sequentially up the memory map. For example the register at \$FF00 is the same as the one at \$FF04 is the same as the one at \$FF08 etc.

PIA 0 is mainly used for handling the keyboard and joysticks ( see page 162 for an example )

PIA 1 has a variety of uses - the six bit number for the

- D - A converter.
- the printer strobe.
- cassette input.
- controls the VDG.
- single bit sound.

#### DETAILS OF THE PIA'S ON THE DRAGON

ADDRESS	DESCRIPTION
---------	-------------

\$FF00 PDROA	Keyboard row input after selecting the row to be examined with \$FF02 (see below). The information returned is 1's where a key is not pressed and 0's where it is.
--------------	--

\$FF01 CROA    b0,b1 - Control of the horizontal sync clock  
                   b2 - DDR (0) / PDR (1)  
                   b3 - SEL1 - LSB of analogue MUX select lines.  
                   b4,b5 - 1 always  
                   b6 - not used  
                   b7 - horizontal interrupt flag

\$FF02 PDROB Keyboard row select. Send all 1's except for a  
                   0 in the bit corresponding to the row to be  
                   examined

\$FF03 CROB    b0,b1 - control of field sync clock  
                   b2 - DDR (0) / PDR (1)  
                   b3 - SEL2 MSB of analogue MUX select lines  
                   b4,b5 - 1 always  
                   b6 - not used  
                   b7 - field sync interrupt flag

\$FF20 PDR1A    b0 - cassette input  
                   b1 - printer strobe out  
                   b2-b7 - A - D input

\$FF21 CR1A    b0 - printer acknowledge.  
                   b1 - unused.  
                   b2 - DDR (0) / PDR (1)  
                   b3 - cassette motor control ON (1) / OFF (0)  
                   b4,b5 - 1 always  
                   b6 - not used  
                   b7 - CD interrupt flag

\$FF22 PDR1B    b0 - printer busy  
                   b1 - single bit sound  
                   b2 - RAM size 0 = 4K, 1 = 16K  
                   b3-b7 - VDG control lines

\$FF23 CR1B    b0,b1 - control of cartridge interrupts  
                   b2 - DDR (0) / PDR (1)  
                   b3 - six bit sound enable  
                   b4,b5 - 1 always  
                   b6 - not used  
                   b7 - cartridge interrupt flag

## Screen Memory

The Dragon has a "memory-mapped" display. This means that there is a block of bytes somewhere in memory, which specifies the contents of the screen. Each byte holds the data that determines the dot pattern that is displayed at the character location that corresponds to that byte. For instance, the normal text screen is located in memory locations \$400-\$5FF. The byte at location \$400 determines which character is in the top left corner of the screen ( See appendix F for the codes for the characters ). The byte at \$401 specifies the character in the second position of the first line. This continues until we get to the last character in the line, which is specified by the byte at \$41F. Location \$420 contains the first character of the second line, and so on.

Both the text and semigraphics modes 4 and 6 work on this basis, with one byte for each character position. Semigraphics modes 8, 12, and 24 are peculiar modes requiring 4, 6, and 12 bytes for each character position. If you are really enthusiastic, see appendix B, but these modes are rarely used. The full graphics modes are the most interesting.

In the full graphics modes you have control over individual pixels (short for picture elements). A pixel is the smallest dot that can be addressed on the screen and they have different sizes in different modes, the range of different colours each pixel can have is also determined by the mode. If you have small pixels, you can make more detailed pictures. However more pixels are needed, so more memory is required. These pixels can be in either two colours or 4 colours. One bit is enough to specify the colour of a pixel in two-colour mode. Therefore, in one byte you can fit 8 pixels ( see appendix B ). The first byte of screen memory specifies the colours of the first 8 pixels in the first row of pixels. The next byte specifies the colours of the next 8 pixels in the first row, and so on. Two bits are required to specify the colour of a pixel in 4 colour mode. Therefore, only four pixels fit in each byte giving pixels twice as wide i.e. half the resolution. The colour of each pixel is specified by a pair of bits in the byte ( see appendix B ).

## The Hardware

The graphics display on the Dragon is controlled by two chips. The first is an MC6847 Video Display Generator ( or VDG for short ! ). This chip reads the screen memory and interprets it. It creates a signal for the T.V. set which will display the picture. Access to this chip is through the B data port of PIA 2 ( location DEVPIA=\$FF22 ). The 5 most significant bits ( bits 3 to 7 ) are connected to the VDG. Bit 3 specifies the colour set ( 0 for colour set 0, 1 for colour set 1, sensibly enough ! ). Bits 4 to 7, ( together with the display mode register of the SAM described below )

specify the mode.

The other chip is an MC6883 Synchronous Address Multiplexer (or SAM). This chip generates timing signals, and controls accesses to the dynamic RAM. It shares the RAM between the CPU and the VDG. It contains various registers with a total of 16 bits. These are set one bit at a time, by writing to various locations in memory. Bit 0 is cleared by writing any byte to address DEVSAM ( address \$FFC0 ). It is set by writing to address DEVSAM+1 ( address \$FFC1 ). Bit 1 is cleared (set) by writing to address DEVSAM+2 (DEVSAM+3), and so on up to bit 15, which is cleared (set) by writing to address DEVSAM+30 (DEVSAM+31). Bits 0 to 2 help to control the display mode, and are referred to as the display mode register. Bits 3 to 9 specify the start location of the screen currently being displayed ( in multiples of 512 bytes ), and are referred to as the page select register.

Both devices have to be correctly set up to display a graphics screen. The following is a table of the values required for the main modes.

mode	resolution	colours	size	DMR	VDG
text/SG4	64 x 32	9*	512	0	0
SG6	64 x 48	4*	512	0	1
SG8	64 x 64	9*	2048	2	0
SG12	64 x 96	9*	3072	4	0
SG24	64 x 192	9*	6144	6	0
CG	64 x 64	4	1024	1	8
CG	128 x 64	2	1024	1	9
CG	128 x 64	4	2048	2	10
CG mode 0	128 x 96	2	1536	3	11
CG mode 1	128 x 96	4	3072	4	12
CG mode 2	128 x 192	2	3072	5	13
CG mode 3	128 x 192	4	6144	5	14
CG mode 4	256 x 192	2	6144	6	15

mode: SG = semigraphics

CG = full graphics

colours : 9\* = 8 foreground colours on black background

size : number of bytes required

DMR : display mode register contents

VDG : VDG control (bits 4 to 7 of PIA2B)

See appendix B for a more complete description of each mode.

## The Use of the Direct Page

Since the direct page is the most convenient place to put variables, I have used it quite a lot. This causes some problems, since BASIC also uses the direct page ( It's no fool ! ). Fortunately, the 6809 allows the direct page to be any page (256 bytes) in memory. BASIC uses page 0 as its direct page. If you mess about with page 0 locations, you won't be able to return to BASIC once you've finished. Also

some are used by ROM routines which you might want to use. The programs change the direct page so that it won't clash with BASIC. However, if you want to return to BASIC, or use the ROM routines, you have to restore the direct page register to 0 beforehand.



## Program: PIA keys

There are two basic ways to scan the keyboard in a machine language program; use the ROM routine POLCAT (\$BBE5), or directly from the PIA. This program does just that.

### HOW IT WORKS

The first thing the program does is initialize the PIA. To read the keyboard DDRA needs to have all its bits as input while DDRB ( see the section on the PIA if you dont know what I am talking about) needs to have all its bits as output (why this is so is explained later). The next step is to set up the screen to show the user which key is being pressed. This is done by displaying a matrix of keys and flashing the one(s) being pressed. Because the DRAGON has no screen characters to represent some keys (ENTER, BREAK, CLEAR, etc.) some of the special characters are used.

After the screen displays all the goodies the real program starts up. To read the keyboard using the PIA is a little tricky. The keyboard is set up in a matrix (same as on the screen) and only one row at a time can be looked at. The row currently being read is selected by outputting to the PDRB and the row is read in on the PDRA. When selecting the row which is to be look at, STore a number in the PDRB with all 1's in it except for a 0 in the bit corresponding to the row you wish to examine. After SToring this number another number can be read from the PDRA which, in turn, has all 1's in it except for a 0 in the bit where a key is pressed.

This program sets up a mask which is rotated through so that each row is looked at. When each row is read the row itself is shifted through and each bit checked to see if that key is pressed. In your own programs this is probably not necessary as you just need to compare the row with a mask to check if the particular key you are after is pressed i.e. down arrow, BREAK, etc.

### HOW TO RUN THE PROGRAM

Just type in the program (using an assembler, loader or monitor), after reserving space for it and EXEC it. The screen should clear and the matrix and legend appear, after this nothing happens until you press a key. When a key is pressed the corresponding character on the screen flashes. More than one key can be pressed at a time and the more you press the faster the characters flash. Note that if you press three keys which happen to form three corners of a rectangle then the fourth corner flashes also. This is a hardware problem and cannot be fixed by programing ( the DRAGON's BASIC allows only one key from each column to be pressed at a time, in order to prevent this problem from arising ).

\* FILE-PIA

\*

\* SCANS KEYBOARD DIRECTLY FROM PIA

\* AND DISPLAYS KEYS PRESSED

\*

0001	OE00		ORG	\$4000	
0002	FF00	PIA	EQU	\$FF00	Address of PIA
0003	0400	SCREEN	EQU	\$400	Screen address
0004	4000	207D	BRA	LAB1	Go to program
0005	4002	NROLC	RMB	1	Column counter
0006	4003	NROLR	RMB	1	Mask counter

\* Screen character matrix

0007	4004	6063	CSET	FCB	\$60,\$63
0008	4006	58504840		FCC	/XPH@/
0009	400A	78706064		FCB	\$78,\$70,\$60,\$64
0010	400E	59514941		FCC	/YQIA/
0011	4012	79716065		FCB	\$79,\$71,\$60,\$65
0012	4016	5A524A42		FCC	/ZRJB/
0013	401A	7A7260605E		FCB	\$7A,\$72,\$60,\$60,\$5E
0014	401F	534B43		FCC	/SKC/
0015	4022	7B73606061		FCB	\$7B,\$73,\$60,\$60,\$61
0016	4027	544C44		FCC	/TLD/
0017	402A	6C7460605F		FCB	\$6C,\$74,\$60,\$60,\$5F
0018	402F	554D45		FCC	/UME/
0019	4032	6D7560607F		FCB	\$6D,\$75,\$60,\$60,\$7F
0020	4037	564E46		FCC	/VNF/
0021	403A	6E76606660		FCB	\$6E,\$76,\$60,\$66,\$60
0022	403F	574F47		FCC	/WOG/
0023	4042	6F77		FCB	\$6F,\$77

\* Messages for legend

0024	4044	6160	MSG1	FCB	\$61,\$60
0025	4046	444F574E60		FCC	'DOWN', \$60
0026	404B	4152524F57		FCC	'ARROW', 0
0027	4051	7F60	MSG2	FCB	\$7F,\$60
0028	4053	5249474854		FCC	'RIGHT', \$60
0029	4059	4152524F57		FCC	'ARROW', 0
0030	405F	6360	MSG3	FCB	\$63,\$60
0031	4061	454E544552		FCC	'ENTER', 0
0032	4067	6460	MSG4	FCB	\$64,\$60
0033	4069	434C454152		FCC	'CLEAR', 0
0034	406F	6560	MSG5	FCB	\$65,\$60
0035	4071	425245414B		FCC	'BREAK', 0
0036	4077	6660	MSG6	FCB	\$66,\$60
0037	4079	5348494654		FCC	'SHIFT', 0

\*

\* Start of initialisation code

\*

0038	407f	108EFF00	LAB1	LDY	#PIA	Base pointer
------	------	----------	------	-----	------	--------------

\* Set data direction registers

\* Not necessary, but included as an example

0039	4083	A621	DDRA	LDA	1,Y	Set DDRA
------	------	------	------	-----	-----	----------

select bit to 0

0040	4085	84FB		ANDA	##%11111011
------	------	------	--	------	-------------

0041	4087	A721		STA	1,Y
------	------	------	--	-----	-----

0042 4089 6FA4		CLR ,Y	PDR all input
0043 408B 8A04		ORA #00000100	Set selstc bit
to 1			
0044 408D A721		STA 1,Y	
0045 408F A623	DDR8	LDA 3,Y	Set DDR8 select
bit to 0			
0046 4091 84FB		ANDA #011111011	
0047 4093 A723		STA 3,Y	
0048 4095 C6FF		LDB #0FF	
0049 4097 E722		STB 2,Y	PDR all output
0050 4099 8A04		ORA #00000100	Set select bit
to 1			
0051 409B A723		STA 3,Y	
* Clear the screen			
0052 409D 8E0400		LDX #SCREEN	
0053 40A0 CC6060	CLS	LDD #06060	Two blanks
0054 40A3 ED81	CL1	STD ,X++	Put them on the
screen			
0055 40A5 8C0600		CMPX #0600	Check for end of
screen			
0056 40A8 23F9		BLS CL1	Loop if not
* Display key matrix on screen			
0057 40AA 8E0420		LDX #0420	
0058 40AD CE4004	SMAT	LDU #CSET	Start of matrix
string			
0059 40B0 C608	SLMAT	LDB #8	Offset from
start of line			
0060 40B2 A6C0	SRMAT	LDA ,U+	Get character
0061 40B4 A785		STA B,X	Put it on the
screen			
0062 40B6 CB02		ADDB #2	Move across 2
positions			
0063 40B8 C116		CMPB #22	Line done ?
0064 40BA 23F6		BLS SRMAT	Loop if not done
0065 40BC 8C04E0		CMPX #04E0	Matrix done ?
0066 40BF 2205		BHI MSGS	Exit if done
0067 40C1 308820		LEAX \$20,X	Next line
0068 40C4 20EA		BRA SLMAT	Loop
* Put legend on screen			
0069 40C6 108E0561	MSGS	LDY #0561	Screen
position			
0070 40CA 1F21		TFR Y,X	Copy to X
0071 40CC CE4044		LDU #MSG1	Message
0072 40CF A6C0	MLA1	LDA ,U+	Get 1 byte of
message			
0073 40D1 2704		BEQ NEXT	Exit if 0
0074 40D3 A780		STA ,X+	Put byte to
screen			
0075 40D5 20F8		BRA MLA1	Loop
0076 40D7 31A810	NEXT	LEAY \$10,Y	Next message
position			
0077 40DA 1F21		TFR Y,X	Copy to X
0078 40DC 8C05C1		CMPX #0561+6*\$10	6 messages
done ?			
0079 40DF 25EE		BLO MLA1	Loop if not
* Start of the main program			

0080	40E1	8E0426	PROG	LDX #0426	First
characterin matrix					
0081	40E4	1F12		TFR X,Y	Copy to Y
0082	40E6	C608		LDB #8	Set mask shift
counter					
0083	40E8	F74003		STB NROLR	
0084	4CEB	CEFF00		LDU #PIA	Set PIA base
address					
0085	40EE	86FE		LDA #011111110	Row select mask
0086	40F0	C608	PL1	LDB #8	Set column shift
counter					
0087	40F2	F74002		STB NROLR	
0088	40F5	A742		STA 2,U	Mask PIA
0089	40F7	E6C4		LDB ,U	Reads columns in
row					
0090	40F9	53		COMB	Make it nicer
0091	40FA	270C		BEQ NROW	Branch if no
keys					
0092	40FC	58	PL2	ASLB	Shift column
into carry					
0093	40FD	3002		LEAX 2,X	Move to next
matrix position					
0094	40FF	2402		BCC PL3	Branch if no key
0095	4101	8D14		BSR FLASH	Flash key
0096	4103	7A4002	PL3	DEC NROLR	Shifted 8 times
?					
0097	4106	26F4		BNE PL2	Loop if not
0098	4108	31A820	NROW	LEAY \$20,Y	Move to next row
0099	410B	1F21		TFR Y,X	Copy to X
0100	410D	1A01		SEC	Set carry flag
0101	410F	49		ROLA	Rotate row
select mask					
0102	4110	7A4003		DEC NROLR	Rotated 8 times
?					
0103	4113	26DB		BNE PL1	Loop if not
0104	4115	20CA		BRA PROG	Start again
* Subroutine to flash key					
0105	4117	3406	FLASH	PSHS A,B	Save registers
0106	4119	E684		LDB ,X	Get key
character					
0107	411B	C840		EORB #\$40	Invert it
0108	411D	E784		STB ,X	Put it back
0109	411F	4F		CLRA	Set delay
counter					
0110	4120	4A	PLA	DECA	Looped 256 times
?					
0111	4121	26FD		BNE PL4	If not, loop
again					
0112	4123	C840		EORB #\$40	Restore key
character					
0113	4125	E784		STB ,X	Put it back
0114	4127	3586		PULS A,B,PC	Restore
registers & return					

NO ERRORS FOUND

CL1	40A3	CLS	40A0	CSET	4004	DDRA	4083
DDR8	408F	FLASH	4117	LAB1	407F	MLA1	40CF
MSG1	4044	MSG2	4051	MSG3	405F	MSG4	4067
MSG5	406F	MSG6	4077	MSG8	40C6	NEXT	40D7
NROLC	4002	NROLR	4003	NROW	4108	PIA	FF00
PL1	40F0	PL2	40FC	PL3	4103	PL4	4120
PROG	40E1	SCREEN	0400	SLMAT	40B0	SMAT	40AD
SRMAT	40B2						

## Program: Score

SCORE is a program which maintains and displays a score on a high resolution graphics screen. It is an introduction to using machine language programs to generate high resolution graphics. It contains some useful routines: CONVRT converts coordinates into locations in screen memory ( for modes 1, 3, and 4 ). SCREEN sets the display devices to display a graphics screen. CLEAR is a simple routine for clearing the screen. If you want to run the program right now, skip the next section, and read the HOW TO RUN THE PROGRAM section.

### SCREEN VARIABLES :

I have defined a number of variables on the direct page of memory. These variables are required by most of the routines. They inform the routines of various characteristics of the screen currently being worked on.

START - specifies the location of variables on the direct page of memory. This MUST be a multiple of 512 bytes. The Dragon's first graphics page starts at location \$600.

YMAX - ( 1 byte ) specifies the number of lines of pixels on the screen ( 192 for modes 3 and 4, 92 for mode 1 ).

LOWBIT - is a mask. Bits 3 to 7 of the x coordinate, together with the y coordinate, are used to work out the address of the byte which contains that pixel. Bits 0 to 2 of the x coordinate are used to determine the position of the pixel within that byte. LOWBIT is used to mask off these 3 bits by "ANDing" it with the x coordinate. Since in 4 colour mode each pixel spans two bits, only even addresses make sense. LOWBIT therefore contains %00000110, to force the address even as well.

LENGTH - specifies the length of the screen in bytes. For these routines, this should be 32 x YMAX, since each of the allowed modes ( 1, 3, and 4 ) requires 32 bytes per line. If you want to do odd things like clearing only part of a screen, you could set LENGTH to some lesser number ( see program DEMO ).

SAM - is the value which the display mode register should be set to.

PIA - is the value which should be stored in PIA2 B data register to control the VDG. This number is 16 x the value in the VDG column of the table of modes, + 8 if you want colour set 1 rather than colour set 0.

### COORDINATES :

In these programs, a pixel on the screen is referred to by its coordinates. The top left corner of the screen always has

coordinates XCOORD=0, YCOORD=0. The point in the bottom right hand corner has coordinates XCOORD=255, YCOORD=YMAX-1. In modes 3 and 4, where there are 192 lines of pixels on the screen, this is XCOORD=255, YCOORD=191, just the same as BASIC. In mode 1, where there are 96 lines of pixels on the screen, this is XCOORD=255, YCOORD=95. Note that in the 4 colour modes, two adjacent pixels differ by 2 in their x coordinates.

#### HOW TO RUN THE PROGRAM :

Allocate space for the program using CLEAR. Load the assembled program into memory. Then run it using EXEC. You should see a clear screen with a six digit number in the top left corner of the screen. The number should be constantly increasing. Reset the machine to stop the program.

#### HOW TO USE THE ROUTINES IN YOUR OWN PROGRAM :

##### SCREEN

This routine sets the Dragon's display chips ( the SAM and VDG ) to display the screen specified by the screen variables ( see above ). It is written in PIC ( position independent code ), so it can be relocated anywhere. Set the screen variables START, SAM, and PIA as described above. Then call SCREEN ( using a BSR SCREEN, LBSR SCREEN, or JSR SCREEN ). The change of screen may occur in the middle of the VDG sending the old screen to the T.V. . If this happens you will see a flicker. To stop this, include a SYNC instruction before the call ( This will synchronise the program with the VDG, so that the change will not occur while the old screen is being displayed ).

##### CLEAR

This routine clears the screen described by the screen variables. It is written in PIC and so can be relocated. Set the values of START and LENGTH as appropriate. Put the pair of bytes you wish to fill the screen with in register D ( Usually \$0,\$5555, \$AAAA, or \$FFFF to set the screen to colour 0, 1, 2, or 3 respectively ). Call CLEAR

##### CONVRT

This routine converts a pair of coordinates ( B = x coordinate, A = y coordinate ) into two values : the address of the byte containing that point ( in X ), and a bit pattern ( in B ). This bit pattern has one bit set. This bit specifies where, in that byte, the pixel lies. For example, imagine a point has coordinates (4,0) ( ie. A=0, B=4 ). After calling CONVRT, X will contain the value of START ( since the first 8 points are contained in the first byte ). B will be %00001000 ( the 5th bit from the left is set, since (4,0) is the 5th point in that group of 8. (0,0) is the first ). CONVRT is written in PIC.

Make sure the screen variables START and LOWBIT are set appropriately. Put the X and Y coordinates in the B and A registers respectively. Call CONVRT.

In two colour mode:

To set that point,

ORB ,X set bit

STB ,X save modification

To reset the point,

COMB complement bit pattern

ANDB ,X clear bit

STB ,X save modification

( see EXPLODE for an example of "EORing" a pixel ).

4 colour graphics is more complicated. Two bits are used to specify the colour : the bit in B and its neighbour to the right. To set a pixel to colour 2 ( blue/magenta ) for example,

STB TEMP save bit

LSRB shift bit right

COMB complement bit pattern

ANDB ,X clear right bit

ORA ,X set left bit

STB ,X save modification

SCADD

Adds two Binary Coded Decimal numbers in memory. This is useful to add points to a score. SCADD is written in PIC. Both numbers must have an even number of digits.

Say you wanted a 6 digit score, which could be increased by 5, 10 or 200 points. You would need four BCD numbers each 3 bytes long :

SCORE FCB \$00,\$00,\$00 start score at 0

SMALL FCB \$00,\$00,\$05 5 point increment

MEDIUM FCB \$00,\$00,\$10 10 point increment

LARGE FCB \$00,\$02,\$00 200 point increment

To add 200 to the score, set U to the address of the score ( LDU #SCORE). Set X to the address of the increment ( LDX #LARGE ). Set B to the number of pairs of digits ( LDB #3 ). Call SCADD, and the score will be changed in memory. Use SCSHOW to display the new score on the screen.

SCSHOW

Displays a score on a high resolution screen. Before using this routine you must work out the position of the score on the screen. You need the address of the byte containing the top line of the first digit. ( You might want to use CONVRT to do this ). Put this address in the X register. Put the address of the score in the U register. Put the number of



pairs of digits in the B register. Then call SCSSHOW. Remember that you will only see the score if you have set the display devices so that this screen will be displayed ( using SCREEN for example ). Be careful when using two screens. You will need to put the score on both.

The table of digits ( DIGTAB ) at the end of the routine can be changed to suit your taste in digits. The only restriction is that the digits are only one byte wide. If you wish to change the length of the digits you need to change line 97 appropriately. This will change the instructions in lines 85 and 89.

```

* FILE-SCORE
*
* ROUTINES TO MAINTAIN A SCORE ON
* A HIGH RES. GRAPHICS SCREEN
*
0001 0E00                ORG $4000

*
* DEVICE ADDRESSES
*
0002 FF01                PIA1CA EQU $FF01        PIA1 control reg
A
0003 FF03                PIA1CB EQU $FF03        PIA1 contral reg
B
0004 FF20                DAC    EQU $FF20        PIA2 data A =
D/A converter
0005 FF22                DEVPIA EQU $FF22        PIA2 data B =
VDG control
0006 FF23                PIA2CB EQU $FF23        PIA2 control reg
B
0007 FFC0                DEVSAM EQU $FFC0        SAM-RAM
controller

0008 4000                END

*
* SCREEN VARIABLES
* in direct page
*
0009 4000                ORG 0
0010 C000                START RMB 2        start of
graphics screen
0011 0002                LENGTH RMB 2        YMAX*32
0012 0004                SAM    RMB 1        value for
display mode reg
0013 0005                PIA    RMB 1        value for PIA
0014 0006                LOWBIT RMB 1        mask for last 3
bits of xcoord.
0015 0007                YMAX   RMB 1        number of lines
in screen
0016 0008                REORG

*
* CONSTANTS FOR SCREEN VARIABLES
*
0017 0600                GSTART EQU $600        Start of
graphics pages

* values of YMAX for modes 1,3 &4
* Note : LENGTH=32*YMAX
0018 0060                YMAX1 EQU 96
0019 00C0                YMAX3 EQU 192

```

```

0020 00C0          YMAX4 EQU 192

* values of SAM for modes 1,3 & 4
0021 0004          SAM1 EQU 4
0022 0006          SAM3 EQU 6
0023 0006          SAM4 EQU 6

* values of PIA for modes 1,3 &4
* add CSET0/1 for colour set 0/1
0024 00C0          PIA1 EQU $C0
0025 00E0          PIA3 EQU $E0
0026 00F0          PIA4 EQU $F0
0027 0000          CSET0 EQU 0
0028 0008          CSET1 EQU 8

* values of LOWBIT for modes 1,3 & 4
* mask for last 3 bits of xcoordinate
* also force xcoord. even if 4 colour
mode
0029 0006          LOWB1 EQU 6
0030 0006          LOWB3 EQU 6
0031 0007          LOWB4 EQU 7

0032 4000          END

*
* SCORE TEST PROGRAM
*
0033 4000          BEGIN EQU *
0034 4000 CCO600          LDD #GSTART
0035 4003 DDO0          STD START
0036 4005 CC1800          LDD #YMAX3*32 SET SCREEN
VARIABLES
0037 4008 DD02          STD LENGTH -FOR MODE 3
0038 400A CCC006          LDD #YMAX3*256+LOWB3
0039 400D 9707          STA YMAX
0040 400F D706          STB LOWBIT
0041 4011 CC06E8          LDD #SAM3*256+PIA3+CSET1
0042 4014 9704          STA SAM
0043 4016 D705          STB PIA
0044 4018 CCG000          LDD #0
0045 401B 1700E0          LBSR CLEAR CLEAR SCREEN
0046 401E 1700B4          LBSR SCREEN SET SCREEN
0047 4021          TEST1 EQU *
* ADD INCREMENT TO SCORE
0048 4021 338D008A          LEAU SCORE,PCR ADDRESS OF SCORE
0049 4025 308D0089          LEAX INCR,PCR ADDRESS OF
INCREMENT
0050 4029 C603          LDB #NPAIRS NUMBER OF PAIRS
OFDIGITS
0051 402B 8D0D          BSR SCADD ADD THEM

* DISPLAY NEW SCORE
0052 402D 8E0600          LDX #SCPOS ADDRESS OF LAST
DIGIT ON SCREEN
0053 4030 338D007B          LEAU SCORE,PCR ADDRESS OF SCORE
0054 4034 C603          LDB #NPAIRS NUMBER OF PAIRS

```

```

OFDIGITS
0055 4036 8D13          BSR SCSHOW          SHOW THE SCORE
0056 4038 20E7          BRA TEST1

* SCADD
*
* Add BCD number at X to score at U
* Does not update score on screen
* Position independent code
* U,X,B modified

0057 403A          SCADD EQU *
0058 403A 33C5          LEAU B,U            MUST START FROM
LAST DIGIT
0059 403C 3085          LEAX B,X            DITTO
0060 403E 1CFE          CLC
0061 4040          SCADD1 EQU *          DECIMALLY ADD A
PAIR OF DIGITS
0062 4040 A6C2          LDA ,-U
0063 4042 A982          ADCA ,-X
0064 4044 19           DAA
0065 4045 A7C4          STA ,U
0066 4047 5A           DECB
0067 4048 26F6          BNE SCADD1
* CHECK HERE FOR CARRY IF NEED BE
0068 404A 39           RTS
* end of SCADD

* SCSHOW
*
* Display score from memory location U,
* of length B bytes, to screen at
address X.
* Position independent code
* A,B,X,U modified

0069 404B A6C4          SCSHOW LDA ,U        TAKE MSD
0070 404D 44           LSRA
0071 404E 44           LSRA
0072 404F 44           LSRA
0073 4050 44           LSRA
0074 4051 8D0E          BSR PUTDIG          DISPLAY DIGIT
0075 4053 3001          LEAX 1,X            MOVE RIGHT
0076 4055 A6C0          LDA ,U+            TAKE LSD
0077 4057 840F          ANDA #$F
0078 4059 8D06          BSR PUTDIG          DISPLAY DIGIT
0079 405B 3001          LEAX 1,X            MOVE RIGHT
0080 405D 5A           DECB
0081 405E 26EB          BNE SCSHOW
0082 4060 39           DIGEND RTS

* DISPLAY DIGIT A AT X
PUTDIG EQU *
0083 4061          PSHS A,B,X,U
0084 4061 3456          LDB #LEN            CALCULATE TABLE
0085 4063 C605

```

OFFSET				
0086	4065	3D	MUL	
0087	4066	338D0013	LEAU DIGTAB,PCR TABLE BASE	
ADDRESS				
0088	406A	33CB	LEAU D,U	PATTERN ADDRESS
0089	406C	C605	LDB #LEN	PUT PATTERN TO
SCREEN				
0090	406E	A6C0	PUTDI1 LDA ,U+	
0091	4070	A784	STA ,X	STORE ON CURRENT
LINE				
0092	4072	A78820	STA 32,X	AND NEXT LINE
0093	4075	308840	LEAX 64,X	MOVE DOWN 2
LINES				
0094	4078	5A	DECB	LOOP UNTIL B=0
0095	4079	26F3	BNE PUTDI1	
0096	407B	35D6	PULS A,B,X,U,PC	RETURN
0097	0005		LEN EQU 5	LENGTH OF EACH
PATTERNIN	BYTES			

\* TABLE OF DIGIT PATTERNS  
DIGTAB EQU \*                      TABLE OF DIGIT

0098	407D		
PATTERNS			
0099	407D	FC	FCB %11111100
0100	407E	CC	FCB %11001100
0101	407F	CC	FCB %11001100
0102	4080	CC	FCB %11001100
0103	4081	FC	FCB %11111100
0104	4082	FO	FCB %11110000
0105	4083	30	FCB %00110000
0106	4084	30	FCB %00110000
0107	4085	30	FCB %00110000
0108	4086	FC	FCB %11111100
0109	4087	FC	FCB %11111100
0110	4088	OC	FCB %00001100
0111	4089	3C	FCB %00111100
0112	408A	CC	FCB %11000000
0113	408B	FC	FCB %11111100
0114	408C	FC	FCB %11111100
0115	408D	OC	FCB %00001100
0116	408E	3C	FCB %00111100
0117	408F	OC	FCB %00001100
0118	4090	FC	FCB %11111100
0119	4091	OC	FCB %00001100
0120	4092	3C	FCB %00111100
0121	4093	CC	FCB %11001100
0122	4094	FC	FCB %11111100
0123	4095	OC	FCB %00001100
0124	4096	FC	FCB %11111100
0125	4097	CO	FCB %11000000
0126	4098	FC	FCB %11111100

0127 4099 OC                   FCB %00001100  
 0128 409A FO                   FCB %11110000  
  
 0129 409B FC                   FCB %11111100  
 0130 409C CO                   FCB %11000000  
 0131 409D FC                   FCB %11111100  
 0132 409E CC                   FCB %11001100  
 0133 409F FC                   FCB %11111100  
  
 0134 40A0 FC                   FCB %11111100  
 0135 40A1 CC                   FCB %11001100  
 0136 40A2 OC                   FCB %00001100  
 0137 40A3 30                   FCB %00110000  
 0138 40A4 30                   FCB %00110000  
  
 0139 40A5 FC                   FCB %11111100  
 0140 40A6 CC                   FCB %11001100  
 0141 40A7 FC                   FCB %11111100  
 0142 40A8 CC                   FCB %11001100  
 0143 40A9 FC                   FCB %11111100  
  
 0144 40AA FC                   FCB %11111100  
 0145 40AB CC                   FCB %11001100  
 0146 40AC FC                   FCB %11111100  
 0147 40AD OC                   FCB %00001100  
 0148 40AE 30                   FCB %00110000

\* end of SCSHOW

\* SCORE STORED HERE IN BCD  
 \* 2 DIGITS/BYTE

0149 40AF 000123               SCORE   FCB \$00,\$01,\$23   SCORE=123  
 DECIMAL

\* AND INCREMENT

0150 40B2 000003               INCR   FCB \$00,\$00,\$03   INCREMENT=3

0151 0003                       NPAIRS EQU 3               SCORE HAS 3  
 PAIRS OF DIGITS

0152 0600                       SCPOS EQU GSTART          SCORE POSITION

\* CONVRT

\*

\* Convert A=y-coordinate,

B=x-coordinate

\* into X=address of byte on screen

\* and B has the corresponding bit set.

\* Position independent code

\* X,B modified

0153 40B5                       CONVRT EQU \*

0154 40B5 3426                   PSHS D,Y                   save D & Y

0155 40B7 44                   LSRA                       shift D right 3

bits

0156 40B8 56                   RORB                       -to get offset

```

from
0157 40B9 44          LSRA          -start of screen
0158 40BA 56          RORB          -
0159 40BB 44          LSRA          -
0160 40BC 56          RORB          -
0161 40BD D300        ADDD START    gives address of
byte
0162 40BF 1F01        TFR D,X      put in X reg
0163 40C1 3506        PULS D      get coordinates
again
0164 40C3 D406        ANDB LCWBIT  select last 3
bits
0165 40C5 318D0004    LEAY CONTAB,PCR get Bth byte
from table
0166 40C9 E6A5        LDB B,Y      -
0167 40CB 35A0        PULS Y,PC    restore Y and
return
0168 40CD 80402010    CCNTAB FCB $80,$40,$20,$10
0169 40D1 08040201    FCB $08,$04,$02,$01
* end of CONVRT
* SCREEN
*
* Displays screen as specified by

screen variables

* Position independent code
* No registers modified

0170 40D5          SCREEN EQU *
0171 40D5 3416        PSHS D,X
0172 40D7 B6FF22      LDA DEVPIA    mask PIA onto
high
0173 40DA 8407        ANDA #7       -5 bits of
DEVPIA
0174 40DC 9A05        ORA PIA       -
0175 40DE B7FF22      STA DEVPIA    -
0176 40E1 9604        LDA SAM       set 3 bits of
0177 40E3 C603        LDB #3       -VDG register to
0178 40E5 8EFFC0      LDX #DEVSAM  -SAM
0179 40E8 8D09        BSR SBITS    -
0180 40EA 9600        LDA START    A=start
address/256
0181 40EC 44          LSRA          A=start
address/128
0182 40ED C607        LDB #7       set 7 bits of
Page Select Reg
0183 40EF 8D02        BSR SBITS    -
0184 40F1 3596        PULS D,X,PC
* SET B BITS OF SAM

0185 40F3          SBITS EQU *
X
0186 40F3 A781        STA ,X++     assume last bit
is zero
0187 40F5 44          LSRA          take last bit
0188 40F6 2402        BCC SBO      branch if it was

```

```

0
0189 40F8 A71F          STA -1,X          it was 1 so fix
it !
0190 40FA 5A           SBO  DECB          loop B times
0191 40FB 26F6          BNE SBITS
0192 40FD 39           RTS

```

\* end of SCREEN

\* CLEAR

\*

\* Fills screen with pairs of bytes

given in D.

\* Note : does not stop if LENGTH is odd

\* Position independent code

\* No registers modified

```

0193 40FE          CLEAR EQU *
0194 40FE 3430      PSHS X,Y
0195 4100 9E00      LDX START          pointer to start
of screen
0196 4102 109E02    LDY LENGTH          number of bytes
to clear
0197 4105 ED81      CLEARX STD ,X++      clear 2 bytes
0198 4107 313E      LEAY -2,Y          loop Y/2 times
0199 4109 26FA      BNE CLEARX
0200 410B 35B0      PULS X,Y,PC

```

\* end of CLEAR

```

0201 410D          END

```

```

0202 410D          END

```

NO ERRORS FOUND

```

BEGIN 4000 CLEAR 40FE CLEARX 4105 CONTAB 40CD
CONVRT 40B5 CSET0 0000 CSET1 0008 DAC FF20
DEVPIA FF22 DEVSAM FFC0 DIGEND 4060 DIGTAB 407D
GSTART 0600 INCR 40B2 LEN 0005 LENGTH 0002
LOWB1 0006 LOWB3 0006 LOWB4 0007 LOWBIT 0006
NPAIRS 0003 PIA 0005 PIA1 00C0 PIA1CA FF01
PIA1CB FFO3 PIA2CB FF23 PIA3 00E0 PIA4 00F0
PUTDI1 406E PUTDIG 4061 SAM 0004 SAM1 0004
SAM3 0006 SAM4 0006 SBO 40FA SBITS 40F3
SCADD 403A SCADD1 4040 SCORE 40AF SCPOS 0600
SCREEN 40D5 SCSHOW 404B START 0000 TEST1 4021
YMAX 0007 YMAX1 0060 YMAX3 00C0 YMAX4 00C0

```



## Program: Explode

EXPLODE is a program which generates explosion effects at random places on the screen. It introduces the use of the Digital to Analogue Converter to produce sound effects ( SETUP and first few lines of SPOTTY ), and pseudo random number generators to add unpredictability to games ( RAND1 and RAND2 ). It uses some of the screen routines of SCORE, including CONVRT, which it uses to "EOR" bits onto the screen at specified coordinates. Skip to the section HOW TO RUN THE PROGRAM if you want to avoid the details.

### PSEUDO RANDOM NUMBER GENERATORS :

A pseudo random number generator is an algorithm which generates a sequence of numbers which look random. They are ( to the computer ) completely predictable, and will always produce the same sequence if started off from the same initial conditions. This has its uses occasionally, as in this program ; EXPLODE is called twice for each explosion. The first time around it "EORs" bits pseudo-randomly onto the screen, but the second time around it starts with the same initial value, so it "EORs" exactly the same bits, setting them back to their original values.

The first generator uses the "linear congruence method". In this method the next number in the sequence is generated from the last value by a formula of the kind :

$$\text{NEXT} = ( A \times \text{LAST} + B ) \text{ mod } C$$

(mod is a function that gives the whole integer remainder after integer division).

Note that if C is a power of two, eg  $2^{16}$ , the mod C part only involves taking the 16 least significant bits. This random number generator repeats every  $2^{16}$  calls, but it tends to be a bit more repetitive than this if you only use a few bits of it at a time.

The second generator produces 32 bit numbers. It works by calculating a single bit as a function of the number ( here we use bit 9 EORed with bit 2 ), then rotating this bit into the most significant bit when the rest of the number is shifted to the right by one bit. This generator repeats every 1,073,215,489 calls ( if you don't believe it, count them ! ), but it will get stuck if the initial value is 0.

The calculations involved to calculate the random dot's position in SPOTTY might seem a bit unnecessary. All this trouble is to make the distribution of the dots denser around the central point, and to get rid of the rectangular looking explosion you would get if the dots were distributed evenly throughout the rectangular area. By adding two equally distributed random numbers together you get a distribution which is weighted more

towards the central value. ( Try it with dice : compare the number of 7's to the number of 2's ). The more numbers you add together, the better. Here, just two are good enough to do the job.

To check for points lying off the screen, the coordinates are temporarily made into signed numbers. The old point X=128, Y=128 is the new origin. Now when the offset is added, the result can be checked for overflow. If an overflow occurs, the real sum of these two numbers is outside the limits -128 and +127, making the point off the screen. The coordinates are then converted back again.

#### THE D/A CONVERTER :

The Dragon has a six bit D/A Converter which is used for cassette i/o, joystick input, and sound generation. To use the D/A Converter to produce sound it is necessary to select it as the source of sound going to the T.V. and to enable sound output. Bit 3 of the PIA1A and bit 3 of the PIA1B control registers select the sound source :

PIA1A control bit 3	PIA1B control bit 3	selects sound from
0	0	D/A converter
1	0	cassette
0	1	cartridge
1	1	not used

Bit 3 of the control register for PIA2B enables sound output from the selected source, when set to 1. So both PIA1A and PIA1B need to have bit 3 of their control bytes reset to zero, while bit 3 of PIA2B's control register must be set to one ( see SETUP as an example of how to do this ). Once this is done any value stored in the top 6 bits of PIA2A ( location \$FF20 ) will be converted to an analogue signal and sent to the T.V. set.

Sound is generated by varying this analogue signal. Regular variations produce musical sounding tones (hopefully), while random variations produce noise ( as in SPOTTY ). Of the two random number generators in this program, RAND2 produces a better sequence of numbers for producing noise.

In this example the amplitude or volume of the noise (the range of values the analogue signal varies over) starts off at a high level ( start of TESTO ). It then rapidly decays to zero, with each iteration of the loop EXPL1 ( 4 lines in the middle of the loop ). This gives an explosion effect.

#### HOW TO RUN THE PROGRAM :

Allocate space for the program using CLEAR. Load the assembled program into memory, and run it using EXEC. You should see a clear screen. Multicoloured explosions will appear on the screen. Each fades away before the next one appears. Reset the machine to stop the program.

#### HOW TO USE THE ROUTINES IN YOUR OWN PROGRAM :

##### SETUP

This routine sets up the Dragon so that anything stored in the D/A converter will be heard on the T.V. . It must be called before trying to use the D/AC. Before a number is stored in the D/AC, it should be "ANDed" with DAMASK, since only the top 6 bits are used by the D/AC.

##### RAND1 & RAND2

These are two separate random number generators. Each is written in PIC, and so is relocatable. They use part of the direct page to store the current numbers. So the direct page register must be set up as for SCORE. The initial value of RAND2N must not be zero, since this will cause RAND2 to lock up and keep returning zero. These routines each return a 16 bit random number in the D register when called.

##### EXPLOD

Simulates an explosion. It is written in PIC, and so is relocatable. However it requires the subroutines SPOTTY, RAND1, RAND2, CONVRT, and SCREEN to be in the same relative positions for the BSR's to work correctly. So, all these routines should be moved as a block.

EXPLOD works by randomly "EORing" spots around the centre of the explosion ( XCOORD, YCOORD ). The spots are bounded by a limit, SIZE, which starts off at SIZINI. Every NSPOTS spots, SIZE is increased by SIZINC, until SIZE is greater than SIZLIM. While this is happening, noise is being put out to the D/AC. The volume starts at the value given in the A register, and decreases by a factor of VDECAY every NSPOTS spots. This produces an explosion effect. The B register contains a delay value which can be used to slow the explosion. Before using the routine you must have called SETUP to enable the sound output. Set the values of SIZINI, SIZINC, SIZLIM, NSPOTS, and VDECAY according to the type of explosion ( experiment with the given values ). Set XCOORD and YCOORD to the centre of the explosion. Give an initial noise volume in A, and some value in B ( 1 for no delay ). Then call EXPLOD.

\* FILE-EXPLODE

\*

\* Program to produce an exposition effect.  
\* EORS spots onto the screen randomly  
\* into a growing boundary while generating  
\* noise using the DAC

0001 0E00                   ORG \$4000

\*

\* DEVICE ADDRESSES

\*

0002 FF01	PIA1CA EQU \$FF01	PIA1 control reg
A		
0003 FF03	PIA1CB EQU \$FF03	PIA1 control reg
B		
0004 FF20	DAC EQU \$FF20	PIA2 data A =
D/A converter		
0005 FF22	DEVPIA EQU \$FF22	PIA2 data B =
VDG control		
0006 FF23	PIA2CB EQU \$FF23	PIA2 control reg
B		
0007 FFC0	DEVSAM EQU \$FFC0	SAM-RAM
controller		

0008 4000                   END

\*

\* SCREEN VARIABLES

\* in direct page

\*

0009 4000	ORG 0	
0010 0000	START RMB 2	start of
graphics screen		
0011 0002	LENGTH RMB 2	YMAX*32
0012 0004	SAM RMB 1	value for
display mode reg		
0013 0005	PIA RMB 1	value for PIA
0014 0006	LOWBIT RMB 1	mask for last 3
bits of xcoord.		
0015 0007	YMAX RMB 1	number of lines
in screen		
0016 0008	REORG	

\*

\* CONSTANTS FOR SCREEN VARIABLES

\*

0017 0600	GSTART EQU \$600	Start of
graphics pages		

\* values of YMAX for modes 1

3 & 4

\* Note

0018 0060	YMAX1 EQU 96	
-----------	--------------	--

0019 00C0                   YMAX3 EQU 192  
0020 00C0                   YMAX4 EQU 192

\* values of SAM for modes 1  
3 & 4

0021 0004                   SAM1 EQU 4  
0022 0006                   SAM3 EQU 6  
0023 0006                   SAM4 EQU 6

\* values of PIA for modes 1  
3 & 4

\* add CSET0/1 for colour set 0/1  
0024 00C0                   PIA1 EQU \$CO  
0025 00E0                   PIA3 EQU \$EO  
0026 00F0                   PIA4 EQU \$FO  
0027 0000                   CSET0 EQU 0  
0028 0008                   CSET1 EQU 8

\* values of LOWBIT for modes 1  
3 & 4

\* mask for last 3 bits of xcoordinate  
\* also force xcoord. even if 4 colour mode  
0029 0006                   LOWB1 EQU 6  
0030 0006                   LOWB3 EQU 6  
0031 0007                   LOWB4 EQU 7

0032 4000                   END

\*

\* RANDOM NUMBER VARIABLES

\*

0033 4000                   ORG \$30  
0034 0030                   RAND1N RMB 2               random number 1  
0035 0032                   RAND2N RMB 4               random number 2  
MUST NOT be 0

\*

\* VARIABLES FOR SPOTTY

\*

0036 0036                   SDELAY RMB 1               DELAY QUANTITY  
0037 0037                   VOLUME RMB 1               SOUND VOLUME  
0038 0038                   SIZE RMB 1                 SIZE OF SPOT  
BOUNDARY  
0039 0039                   YCOORD RMB 1  
0040 003A                   XCOORD RMB 1  
0041 003B                   TEMP RMB 2  
  
0042 003D                   SIZINI RMB 1               INITIAL SIZE OF  
EXPLOSION  
0043 003E                   SIZLIM RMB 1               LIMIT TO SIZE OF  
EXPLOSION  
0044 003F                   SIZINC RMB 1               INCREMENT IN  
SIZE OF EXPLOSION  
0045 0040                   NSPOTS RMB 2               NUMBER OF SPOTS

AT EACH SIZE		
0046 0042	VDECAY RMB 1	RATE OF DECAY
OFVOLUME		
0047 0043	REORG	
0048 4000	BEGIN EQU *	
0049 4000 8604	LDA # \$04	DIRECT PAGE =
TEXT SCREEN		
0050 4002 1F8B	TFR A	
DP		
0051 4004 CC0460	LDD # \$0460	
0052 4007 973D	STA SIZINI	INITIAL SIZE
0053 4009 973F	STA SIZINC	SIZE INCREMENT
0054 400B D73E	STB SIZLIM	LIMIT SIZE
0055 400D CC003C	LDD #60	
0056 4010 DD40	STD NSPOTS	SPOTS PER SIZE
0057 4012 86E0	LDA # \$E0	
0058 4014 9742	STA VDECAY	VOLUME DECAY
RATE		
0059 4016 CCC006	LDD #YMAX3*256+SAM3 SET	
SCREENVARIABLES		
0060 4019 9707	STA YMAX	-FOR MODE 3
0061 401B B740D4	STA RAND2	MAKE SURE RAND2
IS NOT ZERO		
0062 401E D704	STB SAM	
0063 4020 86E8	LDA #PIA3+CSET1	COLOUR SET 1
0064 4022 9705	STA PIA	
0065 4024 8607	LDA #LOWB4	ALLOW ODD
X-COORDS		
0066 4026 9706	STA LOWBIT	-TO GET
MULTICOLOURED EFFECT		
0067 4028 CC060C	LDD #GSTART	
0068 402B DD00	STD START	
0069 402D CC1800	LDD #YMAX3*32	
0070 4030 DD02	STD LENGTH	
0071 4032 CC0000	LDD #0	CLEAR SCREEN
0072 4035 1700F7	LBSR CLEAR	
0073 4038 1700CB	LBSR SCREEN	DISPLAY SCREEN
0074 403B 170100	LBSR SETUP	ENABLE SOUND
0075 403E	LOOP EQU *	
0076 403E 170093	LBSR RAND2	INITIAL COORDS
0077 4041 9107	CMPA YMAX	CHECK IF VALID
0078 4043 24F9	BHS LOOP	
0079 4045 DD39	STD YCOORD	& XCOORD
0080 4047 CCFF80	LDD # \$FF80	INITIAL
AMPLITUDE & DELAY		
0081 404A 8D07	BSR EXPLOD	PUT ON SPOTS
SLOWLY		
0082 404C CC0001	LDD # \$0001	TAKE THEM OFF
QUICKLY		
0083 404F 8D02	BSR EXPLOD	-AND SILENTLY
0084 4051 20EB	BRA LOOP	

\* EXPLOD  
 \*  
 \* Subroutine to simulate an explosion.  
 \* Spots are placed randomly onto the screen  
 \* around the centre of the explosion ( at  
 \* XCOORD YCOORD ).

within a boundary.

\* The boundary increases in size from SIZINI  
 \* to SIZLIM in steps of SIZINC.  
 \* The volume of the noise starts at A

\* and decays at a rate given by VDECAY.  
 \* B determines the speed of the explosion.

```

0085 4053          EXPLOD EQU *
0086 4053 9737          STA VOLUME          STORE INITIAL
VOLUME
0087 4055 D736          STB SDELAY          STORE DELAY
CONSTANT
0088 4057 0F30          CLR RAND1N          INITIALISE
RANDOM NUMBER
0089 4059 0F31          CLR RANDIN+1
0090 405B 963D          LDA SIZINI          SET INITIAL SIZE
0091 405D 9738          STA SIZE
0092 405F 109E40       EXPL1 LDY NSPOTS          NUMBER OF SPOTS
TO DO
0093 4062          EXPL2 EQU *
0094 4062 D636          LDB SDELAY
0095 4064 8D16          BSR SPOTTY          PUT ON ONE SPOT
0096 4066 313F          LEAY -1
Y          LOOP UNTIL Y=0
0097 4068 26F8          BNE EXPL2
0098 406A 9637          LDA VOLUME          DECREASE VOLUME
OF NOISE
0099 406C D642          LDB VDECAY
0100 406E 3D          MUL
0101 406F 9737          STA VOLUME
0102 4071 963F          LDA SIZINC          INCREASE SIZE
0103 4073 9B38          ADDA SIZE
0104 4075 9738          STA SIZE
0105 4077 913E          CMPA SIZLIM          CHECK IF BIG
ENOUGH
0106 4079 23E4          BLS EXPL1
0107 407B 39          RTS

```

\* SPOTTY  
 \*  
 \* Subroutine to EOR on spot randomly  
 \* onto the screen  
 \* no further than SIZE  
 \* away from the centre XCOORD  
 \* YCOORD.  
 \* Random noise is produced  
 \* with volume  
 \* given by VOLUME.

0108 407C	SPOTTY EQU *	
0109 407C 5A	DECB	DELAY FOR A
WHILE		
0110 407D 26FD	BNE SPOTTY	
0111 407F 8D53	BSR RAND2	MAKE NOISE
0112 4081 9637	LDA VOLUME	OF CORRECT
VOLUME		
0113 4083 3D	MUL	
0114 4084 84FC	ANDA #DAMASK	AND PUT TO DAC
0115 4086 B7FF20	STA DAC	
0116 4089 8D37	BSR RAND1	GET RANDOM
NUMBER		
0117 408B 9B31	ADDA RAND1N+1	ADD TWO RANDOM
BYTES		
0118 408D 46	RORA	-AND HALVE(GIVES
A MORE CENTRAL		
0119 408E D638	LDB SIZE	-DISTRIBUTION)
THEN SCALE TO SIZE.		
0120 4090 3D	MUL	RANGE 0 TO
SIZE-1		
0121 4091 58	ASLB	
0122 4092 49	ROLA	RANGE 0 TO
2(SIZE-1)		
0123 4093 9038	SUBA SIZE	RANGE -(SIZE-1)
TO SIZE-1		
0124 4095 973B	STA TEMP	SAVE IT
0125 4097 8D29	BSR RAND1	DO THE SAME FOR
X COORD		
0126 4099 9B31	ADDA RAND1N+1	
0127 409B 46	RORA	
0128 409C D638	LDB SIZE	
0129 409E 3D	MUL	
0130 409F 58	ASLB	
0131 40A0 49	ROLA	
0132 40A1 9038	SUBA SIZE	
0133 40A3 973C	STA TEMP+1	
0134 40A5 DC39	LDD YCOORD	CONVERT YCOORD
TO SIGNED NUMBER		
0135 40A7 8080	SUBA #\$80	
0136 40A9 9B3B	ADDA TEMP	SO THAT WE CAN
CHECK		
0137 40AB 2914	BVS NOGO	-FOR OVERFLOW
OFF SCREEN		
0138 40AD 8B80	ADDA #\$80	CONVERT IT BACK
TO UNSIGNED		
0139 40AF 9107	CMPA YMAX	CHECK IF OFF
BOTTOM		
0140 40B1 240E	BHS NOGO	
0141 40B3 C080	SUBB #\$80	DO THE SAME FOR
XCOORD		
0142 40B5 DB3C	ADDB TEMP+1	
0143 40B7 2908	BVS NOGO	



```

0144 40B9 CB80          ADDB #$80
0145 40BB 8D29          BSR CONVRT          CONVERT TO
ADDRESS
0146 40BD E884          EORB
X          PUT SPOT
0147 40BF E784          STB
X
0148 40C1 39           NOGO RTS

```

\* RANDOM NUMBER GENERATORS

\* RAND1

\*

\* Calculates new 16 bit pseudo random number  
RAND1N

\* from its old value  
returning it in D

\* The sequence of numbers repeats every 2\*16 calls

\* Position independent code

\* D modified

```

0149 40C2          RAND1 EQU *
RAND1N=RAND1N*$605+13849 MOD 65536
0150 40C2 DC30          LDD RAND1N          RAND1N
0151 40C4 9B31          ADDA RAND1N+1      RAND1N*$101
0152 40C6 58           ASLB              RAND1N*$202
0153 40C7 49           ROLA              -
0154 40C8 9B31          ADDA RAND1N+1      RAND1N*$302
0155 40CA 58           ASLB              RAND1N*$604
0156 40CB 49           ROLA              -
0157 40CC D330          ADDD RAND1N        RAND1N*$605
0158 40CE C33619        ADDD #13849
0159 40D1 DD30          STD RAND1N
0160 40D3 39           RTS
* end of RAND1

```

\* RAND2

\*

\* Calculates new 32 bit pseudo random number  
RAND2N

\* from its old value  
returning its 2 high bytes in D

\* The sequence of numbers repeats every 1

073

215

489 call

s

\* RAND2N=RAND2N shifted right

\* with MSBit = bit9 eor bit2

\* Note

\* Position independent code

\* D modified

```

0161 40D4          RAND2 EQU *
0162 40D4 9635          LDA RAND2N+3
0163 40D6 46           RORA

```

```

0164 40D7 9834          EORA RAND2N+2
0165 40D9 46           RORA
0166 40DA 46           RORA
0167 40DB 0632        ROR RAND2N
0168 40DD 0633        ROR RAND2N+1
0169 40DF 0634        ROR RAND2N+2
0170 40E1 0635        ROR RAND2N+3
0171 40E3 DC32        LDD RAND2N
0172 40E5 39          RTS
* end of RAND2

```

```
* CONVRT
```

```
*
```

```
* Convert A=y-coordinate
```

```
B=x-coordinate
```

```
* into X=address of byte on screen
```

```
* and B has the corresponding bit set.
```

```
* Position independent code
```

```
* X
```

```
B modified
```

```

0173 40E6          CONVRT EQU *
0174 40E6 3426     PSHS D
Y          save D & Y
0175 40E8 44      LSRA          shift D right 3
bits
0176 40E9 56      RORB          -to get offset
from
0177 40EA 44      LSRA          -start of screen
0178 40EB 56      RORB          -
0179 40EC 44      LSRA          -
0180 40ED 56      RORB          -
0181 40EE D300    ADDD START    gives address of
byte
0182 40F0 1F01    TFR D
X          put in X reg
0183 40F2 3506    PULS D          get coordinates
again
0184 40F4 D406    ANDB LOWBIT    select last 3
bits
0185 40F6 318D0004 LEAY CONTAB
PCR get Bth byte from table
0186 40FA E6A5    LDB B
Y          -
0187 40FC 35A0    PULS Y
PC          restore Y and return
0188 40FE 80402010 CONTAB FCB $80
$40
$20
$10
0189 4102 08040201 FCB $08
$04
$02
$01
* end of CONVRT

```

\* SCREEN

\*

\* Displays screen as specified by screen variables

\* Position independent code

\* No registers modified

```
0190 4106          SCREEN EQU *
0191 4106 3416    PSHS D
X
0192 4108 B6FF22    LDA DEVPIA      mask PIA onto
high
0193 410B 8407     ANDA #7         -5 bits of
DEVPIA
0194 410D 9A05     ORA PIA          -
0195 410F B7FF22    STA DEVPIA       -
0196 4112 9604     LDA SAM          set 3 bits of
0197 4114 C603     LDB #3          -VDG register to
0198 4116 8EFFC0    LDX #DEVSAM      -SAM
0199 4119 8D09     BSR SBITS       -
0200 411B 9600     LDA START       A=start
address/256
0201 411D 44       LSRA            A=start
address/128
0202 411E C607     LDB #7         set 7 bits of
Page Select Reg
0203 4120 8D02     BSR SBITS       -
0204 4122 3596     PULS D
X
PC
* SET B BITS OF SAM
0205 4124          SBITS EQU *      set B bits from
X
0206 4124 A781     STA
X++          assume last bit is zero
0207 4126 44       LSRA            take last bit
0208 4127 2402     BCC SBO         branch if it was
0
0209 4129 A71F     STA -1
X          it was 1 so fix it !
0210 412B 5A       SBO  DECB          loop B times
0211 412C 26F6     BNE SBITS
0212 412E 39       RTS
* end of SCREEN
```

\* CLEAR

\*

\* Fills screen with pairs of bytes given in D

\* Note

\* Position independent code

\* No registers modified

```
0213 412F          CLEAR EQU *
0214 412F 3430    PSHS X
Y
0215 4131 9E00    LDX START       pointer to start
of screen
```

```

0216 4133 109E02          LDY LENGTH      number of bytes
to clear
0217 4136 ED81          CLEARX STD
X++          clear 2 bytes
0218 4138 313E          LEAY -2
Y          loop Y/2 times
0219 413A 26FA          BNE CLEARX
0220 413C 35B0          PULS X
Y
PC
* end of CLEAR

```

```

0221 413E          END
* end of EXPLOD
* SETUP
*
* Enable sound output from the DAC
* Must be called before using the DAC
* Position independent code
* No registers modified

```

```

0222 413E          SETUP EQU *
0223 413E 3406          PSHS D
0224 4140 CCB435          LDD #$B435      set sound output
from DAC
0225 4143 B7FF01          STA PIA1CA      -disable 64
microsecond int.
0226 4146 F7FF03          STB PIA1CB      -and enable 50Hz
int.
0227 4149 863F          LDA #$3F        Enable DAC sound
0228 414B B7FF23          STA PIA2CB      -
0229 414E 3586          PULS D
PC
* end of SETUP

```

```

* mask for data to store to DAC
* AND data with DAMASK befor storing to DAC
0230 00FC          DAMASK EQU %11111100

```

```

0231 4150          END

```

```

0232 4150          END

```

NO ERRORS FOUND

```

BEGIN 4000 CLEAR 412F CLEARX 4136 CONTAB 40FE
CONVRT 40E6 CSET0 0000 CSET1 0008 DAC FF20
DAMASK 00FC DEVPIA FF22 DEVSAM FFC0 EXPL1 405F
EXPL2 4062 EXPLOD 4053 GSTART 0600 LENGTH 0002
LOOP 403E LOWB1 0006 LOWB3 0006 LOWB4 0007
LOWBIT 0006 NOGO 40C1 NSPOTS 0040 PIA 0005
PIA1 00C0 PIA1CA FF01 PIA1CB FF03 PIA2CB FF23
PIA3 00E0 PIA4 00F0 RAND1 40C2 RAND1N 0030
RAND2 40D4 RAND2N 0032 SAM 0004 SAM1 0004
SAM3 0006 SAM4 0006 SBO 412B SBITS 4124
SCREEN 4106 SDELAY 0036 SETUP 413E SIZE 0038

```

SIZINC 003F    SIZINI 003D    SIZLIM 003E    SPOTTY 407C  
START 0000    TEMP 003B    VDECAY 0042    VOLUME 0037  
XCOORD 003A    YCOORD 0039    YMAX 0007    YMAX1 0060  
YMAX3 00C0    YMAX4 00C0

## Program: Music

MUSIC, as the name suggests, is a program to play music, one note at a time. The waveform is variable, and there is limited control over the envelope of the wave ( adjustable decay rate ). The program expands the use of the D/A converter as described in the EXPLODE program. As usual if the details bore you, skip to the HOW TO RUN THE PROGRAM section.

### THE SOUND GENERATION ROUTINE :

Values from the waveform table are put out through the D/A converter at a variable rate. POINT is a 24 bit variable, of which 21 bits are used. Bits 0 to 4 of the most significant byte are an index into the waveform table. This index is repeatedly updated every 64 clock cycles, by adding `FREQ` to `POINT`. The larger the value of `FREQ` is, the faster the index will change. In fact the frequency of the output signal is proportional to `FREQ`.

```
bits : 22221111 111111
       32109876 54321098 76543210
FREQ : 00000000 10000000 00000000
POINT : 00000C11 00000000 00000000
```

```
-----
      index
```

At this stage the index into waveform table is 3, ie. the third value in the table will be sent to the DAC. Every second time `FREQ` is added to `POINT`, the most significant byte of `POINT` will change and a new value will be sent to the DAC. After 64 additions of `FREQ`, bits 16 to 20 of point will have the same value again, and all the values in the waveform table will have been cycled through. Thus a full cycle would be output in 64 loops. Each loop takes 64 clock cycles. The time for a full cycle is therefore 4096 clock cycles. Since the clock speed is 890,000 Hz ( 890,000 cycles per second ), the frequency of the sound would be  $890,000/4096 = 217$  Hz, which is close to A below middle C.

The amplitude of the signal is controlled by the one byte variable, `AMPL`. The entries form the waveform table are multiplied by `AMPL` before being sent to the D/A converter. The start of the waveform table is specified by the X register, while the duration of the tone is specified by the contents of the Y register (  $Y*64$  clock cycles ).

### THE MAIN PROGRAM :

Takes the notes from the score of music and decodes them. It starts the amplitude at its maximum level, and then repeatedly calls `SOUND` while decreasing the amplitude. This produces the decay effect. The length of each note is encoded

in the score with the frequency using the following system :  
Of the 16 bits in the code,  
bits 15 to 12 - specify the octave ( 0 to an approximate  
limit of 5 )  
bits 11 to 8 - specify the note within the given octave (   
0=A to 11=G# )  
bits 7 to 0 - specify the relative duration of the note

Two byte code :

15.14.13.12.11.10. 9. 8. 7. 6. 5. 4. 3. 2. 1. 0.  
(-octave--).(--note---).(-----duration-----).

eg. octave 2, note A#, duration 8 => \$2108

The value \$0000 is used to mark the end of the piece.

Because of the limited number of bits of resolution available  
in the D/AC, and the crude method used to calculate the  
output signal, the signal is not excellent and many harmonics  
can be heard ( especially as a tone decays in intensity to  
near 0 ).

#### WAVEFORMS AND ENVELOPES :

Sound is produced here by sending a sequence of values  
repeatedly to the DAC. This sequence of values is the  
waveform. It determines the "quality" of the sound, whether  
it sounds pure, like a tuning fork, or has more character,  
like an electric guitar. Here the waveform is specified by a  
waveform table. This is a sequence of TLEN=32 bytes.

The waveform has its amplitude (or volume) varied for the  
length of the note. This is called the envelope of the  
waveform. A general form consists of 3 parts : attack,  
sustain, and decay. Consider hitting a piano key. At the  
start of a note, the amplitude increases up to a certain  
value ( attack ). While a note is being held, the amplitude  
may drop a little before levelling out ( sustain ). When the  
note is finished it may take a while for the amplitude of the  
sound to drop to zero ( decay ). In this program, the attack  
is immediate, the sustain doesn't exist, and the decay is  
exponential with a variable rate.

#### HOW TO RUN THE PROGRAM :

Allocate space for the program using CLEAR. Load the  
assembled program into memory, and run it using EXEC. If the  
volume of your T.V. set is turned up, you should hear "Drink  
To Me Only With Thine Eyes". The first half uses a triangular  
waveform. The second half uses a square one.

#### HOW TO USE THE ROUTINES IN YOUR OWN PROGRAM :

##### SOUND

This routine generates a tone of a given frequency,  
amplitude, and waveform for a given length of time. PIC is  
used so the code can be relocated anywhere. Before SOUND is

called, SETUP must be called to enable the sound. The 3-byte frequency is stored in `FREQ`. The one byte amplitude is stored in `AMPL`. The `X` register is made to point to the start of a waveform table. The `Y` register is loaded with a value which determines the duration of the note. When `SOUND` is called the specified note is produced.

By varying the amplitude, frequency, and waveform, many different sound effects can be produced. In this program only the amplitude is varied. Much more than this is possible.

#### PLAY

This routine takes an encoded score of music in memory, and converts it to sound. This routine is written uses `PIC`, but it calls `SOUND` and accesses `NOTAB`. All three must be kept together if `PLAY` is to work properly. `SETUP` must be called to enable the sound output, before using `PLAY`. The address of the waveform table must be stored in `WAVTAB`. The speed of play is determined by the two byte number `SPEED` ( 1 = very fast , `$FFFF` = very slow ). The decay rate is determined by `DECAY` ( 0 = very fast , `$FF` = very slow ) and `SPEED`. Finally the `U` register must point to the start of the music score, which is encoded as described above. Calling `PLAY` plays the music.



\* FILE-MUSIC  
 \*  
 \* SOUND GENERATION PROGRAM  
 \*  
 \* This program, which plays a tune,

illustrates how

\* sound can be produced on the Dragon.

```
0001 0E00                ORG $4000

*
* DEVICE ADDRESSES
*

0002 FF01                PIA1CA EQU $FF01        PIA1 control reg
A
0003 FF03                PIA1CB EQU $FF03        PIA1 control reg
B
0004 FF20                DAC EQU $FF20          PIA2 data A =
D/A converter
0005 FF22                DEVPIA EQU $FF22       PIA2 data B =
VDG control
0006 FF23                PIA2CB EQU $FF23        PIA2 control reg
B
0007 FFC0                DEVSAM EQU $FFC0         SAM-RAM
controller

0008 4000                END
```

\* VARIABLES FOR MUSIC  
 \* in direct page, actually located at

```
$4420
0009 4000                ORG $20
0010 0020                SPEED RMB 2
0011 0022                DECAY RMB 1
0012 0023                WAVTAB RMB 2
0013 0025                FREQ RMB 3
0014 0028                POINT RMB 3
0015 002B                AMPL RMB 1
0016 002C                DURATN RMB 1
0017 002D                REORG

0018 0020                TLEN EQU 32

0019 4000                BEGIN EQU *
0020 4000 3408           PSHS DP
0021 4002 8644           LDA #$44        DIRECT PAGE
NUMBER
0022 4004 1F8B           TFR A,DP        SET DIRECT PAGE
0023 4006 8E0100        LDY #$100
0024 4009 9F20           STX SPEED       SET SPEED
0025 400B 86F0           LDA #$F0
0026 400D 9722           STA DECAY       SET DECAY REATE
0027 400F 308D0086      LEAX TRIANG,PCR TRIANGULAR
```

WAVEFORM		
0028 4013 9F23	STX WAVTAB	
0029 4015 1700C1	LBSR SETUP	ENABLE SOUND
0030 4018 338D00CF	LEAU MUSIC,PCR	U IS POINTER TO
POSITION IN SCORE		
0031 401C 8D0E	BSR PLAY	PLAY IT
0032 401E 308D0097	LEAX SQUARE,PCR	SQUARE WAVE
0033 4022 9F23	STX WAVTAB	
0034 4024 338D00C3	LEAU MUSIC,PCR	
0035 4028 8D02	BSR PLAY	
0036 402A 3588	PULS DP,PC	

\* PLAY  
 \*  
 \* ROUTINE TO PLAY MUSIC  
 \* The score is located at U.  
 \* DECAY holds the decay rate for each

note

\* (0=v.fast - \$FF=v.slow)  
 \* SPEED holds the speed at which the

music is played

\* (1=v.fast - \$FFFF=v.slow )  
 \* note: SPEED also changes the decay

rate

\* The table describing the waveform of

each note

\* is at WAVTAB  
 \* (which is TLEN bytes long)  
 \* Position independent code, but must

include

\* SOUND and NOTAB.  
 \* D,U,X modified

0037 402C	PLAY EQU *	
0038 402C ECC4	LDD ,U	NOTE,DURATION
0039 402E 2733	BEQ PLAYEN	FINISH IF ZERO
0040 4030 D72C	STB DURATN	
0041 4032 840F	ANDA #\$F	NOTE WITHIN
OCTAVE		
0042 4034 308D0049	LEAX NOTAB,PCR	FIND FREQUENCY
FOR THIS NOTE		
0043 4038 48	ASLA	
0044 4039 EC86	LDD A,X	
0045 403B DD26	STD FREQ+1	SAVE FREQUENCY
0046 403D 0F25	CLR FREQ	HIGH BYTE = 0
0047 403F A6C1	LDA ,U++	GET THE NOTE
AGAIN		
0048 4041 0827	PLAYO ASL FREQ+2	ADJUST FREQUENCY
0049 4043 0926	ROL FREQ+1	-DOUBLING FOR
EACH		
0050 4045 0925	ROL FREQ	-OCTAVE
0051 4047 8010	SUBA #16	
0052 4049 24F6	BHS PLAYO	
0053 404B 86FF	LDA #\$FF	INITIAL

```

AMPLITUDE
0054 404D 972B          STA AMPL
0055 404F 9E23          LDX WAVTAB           WAVEFORM TABLE

0056 4051              PLAY2 EQU *           PLAY THIS NOTE
0057 4051 109E20        LDY SPEED
0058 4054 8DOE          BSR SOUND
0059 4056 962B          LDA AMPL             DECREASE
AMPLITUDE
0060 4058 D622          LDB DECAY
0061 405A 3D            MUL
0062 405B 972B          STA AMPL
0063 405D 0A2C          DEC DURATN           LOOP UNTIL
DURATN=0
0064 405F 26F0          BNE PLAY2
0065 4061 20C9          BRA PLAY             GET NEXT NOTE

0066 4063 39            PLAYEN RTS           RETURN

```

```

* SOUND
*
* SOUND GENERATION ROUTINE
* Play a single note with frequency

```

FREQ, waveform pointed  
to

```

* by X, amplitude AMPL, and duration Y
* The loop takes 64 clock cycles to

```

execute

```

* Position independent code
* D modified

```

```

0067 4064              SOUND EQU *
0068 4064 DC29          LDD POINT+1         ADD FREQ TO
POINT
0069 4066 D326          ADDD FREQ+1
0070 4068 DD29          STD POINT+1
0071 406A 9628          LDA POINT
0072 406C 9925          ADCA FREQ
0073 406E 9728          STA POINT
0074 4070 841F          ANDA #TLEN-1        POINT IS INDEX
INTO WAVEFORM TABLE
0075 4072 A686          LDA A,X
0076 4074 D62B          LDB AMPL             MULTIPLY BY
AMPLITUDE
0077 4076 3D            MUL
0078 4077 84FC          ANDA #DAMASK         STORE IN DAC
0079 4079 B7FF20        STA DAC
0080 407C 313F          LEAY -1,Y           DECREMENT Y
0081 407E 26E4          BNE SOUND           LOOP UNTIL Y=0
0082 4080 39            RTS

```

```

* end SOUND
*
* NOTE TO FREQUENCY TABLE
*
* Frequencies of notes 0 to 11 in

```

octave 0

\* Arbitrary scale of form

$F(I)=F(0)*(2(I/12))$   
 0083 4081 2066225324 NOTAB FDB  
 8294,8787,9310,9863,10450,11071  
 0084 408D 2DD2308B33 FDB  
 11730,12427,13166,13949,14778,15657

\* end PLAY

\*  
 \* A WAVEFORM TABLE  
 \* FOR A TRIANGULAR WAVE  
 \*

0085	4099	00102030	TRIANG	FCB	\$00,\$10,\$20,\$30
0086	409D	40506070		FCB	\$40,\$50,\$60,\$70
0087	40A1	8090A0B0		FCB	\$80,\$90,\$A0,\$B0
0088	40A5	C0D0E0F0		FCB	\$C0,\$D0,\$E0,\$F0
0089	40A9	FFF0E0D0		FCB	\$FF,\$F0,\$E0,\$D0
0090	40AD	COB0A090		FCB	\$C0,\$B0,\$A0,\$90
0091	40B1	80706050		FCB	\$80,\$70,\$60,\$50
0092	40B5	40302010		FCB	\$40,\$30,\$20,\$10

\* AND A SQUARE WAVE

0093	40B9	00000000	SQUARE	FCB	0,0,0,0
0094	40BD	00000000		FCB	0,0,0,0
0095	40C1	00000000		FCB	0,0,0,0
0096	40C5	00000000		FCB	0,0,0,0
0097	40C9	FFFFFFF		FCB	\$FF,\$FF,\$FF,\$FF
0098	40CD	FFFFFFF		FCB	\$FF,\$FF,\$FF,\$FF
0099	40D1	FFFFFFF		FCB	\$FF,\$FF,\$FF,\$FF
0100	40D5	FFFFFFF		FCB	\$FF,\$FF,\$FF,\$FF

\* SETUP

\*  
 \* Enable sound output from the DAC  
 \* Must be called before using the DAC  
 \* Position independent code  
 \* No registers modified

0101	40D9		SETUP	EQU	*
0102	40D9	3406		PSHS	D
0103	40DB	CCB435		LDD	#\$B435      set sound output
		from DAC			
0104	40DE	B7FF01		STA	PIA1CA      -disable 64
		microsecond int.			
0105	40E1	F7FF03		STB	PIA1CB      -and enable 50Hz
		int.			
0106	40E4	863F		LDA	#\$3F      Enable DAC sound
0107	40E6	B7FF23		STA	PIA2CB      -
0108	40E9	3586		PULS	D,PC

\* end of SETUP

\* mask for data to store to DAC  
 \* AND data with DAMASK befor storing to

DAC		
0109	00FC	DAMASK EQU %11111100

0110 40EB

END

\* MUSIC SCORE

\*

\* Format 2 bytes per note :

\* 4 bits octave number ( 0=low - about

5=high )

\* 4 bits note number ( 0=A - 11=\$B=G#

)

\* 8 bits duration (\$01=short -

\$FF=long )

\* Drink to Me Only With Thine Eyes

MUSIC EQU \*

0111 40EB

FDB \$910

0112 40EB 0910

FDB \$910

0113 40ED 0910

FDB \$910

0114 40EF 0910

FDB \$A20

0115 40F1 0A20

FDB \$A10

0116 40F3 0A10

FDB \$1010

0117 40F5 1010

FDB \$A10

0118 40F7 0A10

FDB \$910

0119 40F9 0910

FDB \$710

0120 40FB 0710

FDB \$910

0121 40FD 0910

FDB \$A10

0122 40FF 0A10

FDB \$1010

0123 4101 1010

FDB \$510

0124 4103 0510

FDB \$A10

0125 4105 0A10

FDB \$920

0126 4107 0920

FDB \$710

0127 4109 0710

FDB \$560

0128 410B 0560

0129 410D 0000

FDB 0

0130 410F

END

NO ERRORS FOUND

AMPL	002B	BEGIN	4000	DAC	FF20	DAMASK	00FC
DECAY	0022	DEVPIA	FF22	DEVSAM	FFC0	DURATN	002C
FREQ	0025	MUSIC	40EB	NOTAB	4081	PIA1CA	FF01
PIA1CB	FF03	PIA2CB	FF23	PLAY	402C	PLAYO	4041
PLAY2	4051	PLAYEN	4063	POINT	0028	SETUP	40D9
SOUND	4064	SPEED	0020	SQUARE	40B9	TLEN	0020
TRIANG	4099	WAVTAB	0023				

## Program: Demo

DEMO is a program which produces an animated display of a human/robot figure. This is an example of the use of the GET, PUT and PUTC routines. If you just want to run the program go to the section HOW TO RUN THE PROGRAM.

GET, PUT, and PUTC :

These routines are like the BASIC GET and PUT commands, but are faster and conserve memory. Like the basic commands they refer to rectangular areas on the screen. To use them you must set up the variables XCOORD, YCOORD, XSIZE and YSIZE. These specify the coordinates of the top left corner of the rectangle, and the size of the rectangle in coordinate units, not in pixels. ie. if the rectangle is 17 pixels wide in 4 colour mode, its XSIZE is 34 not 17. ( 2 bits are required for each pixel to select between the 4 colours ). Also like BASIC an area of memory needs to be reserved to store the shape within that rectangle. Four 4 colour or eight 2 colour pixels will fit in each byte. If XSIZE' is the smallest multiple of 8 which is less than or equal to XSIZE, then you will have to reserve YSIZE $\times$ XSIZE'/8 bytes ( note that that is XSIZE' not XSIZE ). The address of this area must be passed to the subroutine in the U register.

In the example shown below, R stands for a red pixel on the screen, and G for a green one :

```
R R R R G R R
R R R R R G R
G G G G G G G
R R R R R G R
R R R R G R R
```

With a bit of imagination this shape might look like a decent arrow. If you wanted to store it away so that you could put it to the screen elsewhere, you could use GET ( assuming you already have this on the screen somewhere ). First, set XCOORD and YCOORD to the coordinates of the red pixel in the top left corner. Then set YSIZE to 5, and XSIZE to 14 ( not 7, since each 4 colour pixel is 2 coordinate points wide ). Somewhere in memory you have 10 ( =  $5 \times 16 / 8$  ) bytes reserved to store it. Load the U register with this address, do a JSR GET or BSR GET, and it's done.

To put this shape somewhere else on the screen at some later time, set XCOORD and YCOORD to the new coordinates, and XSIZE and YSIZE to their original values of 14 and 5 respectively. Load the U register with the address of the storage area, and call PUT. PUTC is a version of PUT which takes colour 0 (green/buff) to be transparent.

This messing about with sizes could get a bit tedious. Why not save the size with the shape ? You can, in fact the subroutines SGET, SPUT, and SPUTC in DEMO do this ( they also set the coordinates to the contents of the D register ). They tailor these routines for a particular purpose, making the routines easier to use. ( There are some times when you would not want to save the size with each shape, for example when you have many letter shapes all of the same size. )

The bytes are stored in vertical columns, top to bottom, from right to left. To illustrate, the example above would be

```

stored as
  rightmost column first
byte 0  GRR-
byte 1  RGR-
byte 2  GGG-
byte 3  RGR-
byte 4  GRR-
  then left column
byte 5  RRRR
byte 6  RRRR
byte 7  GGGG
byte 8  RRRR
byte 9  RRRR

```

You should notice that this is DEFINITELY not the order in which they are stored in screen memory. There they are stored in horizontal rows, left to right, top to bottom. Also it is the first ie. rightmost column which is left incomplete if XSIZE is not a multiple of 8.

PUT has to prepare a mask which will remove those areas of the screen which are to be written over. For most of the bytes in the shape this is fairly simple since all 8 bits of the byte are used, but for the ones on the rightmost edge this is not always the case, and a special mask has to be prepared. The easiest way to go about this is to prepare the mask once at the start, and to get all the tricky parts out of the way together. Hence the funny order.

PUT starts off by calling INIT. INIT firstly checks the size of the object to make sure it is non-zero. It then calculates the width of the shape in bytes,  $XSIZE'/8$ , using the fact that  $XSIZE'/8 = (XSIZE+7)/8$ . This value is stored in the outer loop counter XCOUNT. It checks to make sure that the start coordinates are on the screen, then calls CONVRT to calculate the address of the top left screen byte covered by the rectangle. The bit pattern is saved in SHIFT. It will be used later to shift the shape bytes into the correct position in the D register to be able to be masked onto the screen. The mask required for processing the rightmost bytes contains  $(XSIZE \text{ mod } 8)$  one bits in the high end of the byte, with the rest of the bits zero ( or all bits 1 if XSIZE is a multiple of 8 ). In the above example,  $XCOUNT = XSIZE'/8 = 2$ , the mask is %11111100, and if XCOORD=4 say, then  $SHIFT=\%00001000$ . If there was an error somewhere, it removes the return address from the stack and exits the calling routine without doing anything.

PUT takes pairs of bytes from the screen and inserts the shape bytes into the correct position. It then returns the modified pair of bytes to the screen. The mask byte returned from INIT must be shifted into position. This is done by multiplying by SHIFT then shifting it one bit left ( This is done because multiplying by SHIFT effectively shifts the mask by between 1 and 8 bits to the right, where a shift of between 0 and 7 bits is required ). eg %11111100 would get

shifted to %00001111 11000000 in the above example. MASK is actually the complement of this in PUT, ie %11110000 00111111. X is decremented, moving us one byte to the left. Each pair of screen bytes in the rightmost column is ANDed with this value to erase the bits covered by the rectangle. Then the shape byte, which has been shifted into the correct position, is added in before the pair is returned to the screen. Columns to the left are processed in the same way, but all 8 bits are used in these columns, so the initial mask is %11111111.

PUTC works on a similar basis, but the mask is generated for each shape byte individually. Each pair of bits in the mask is either set or reset, depending on whether the corresponding pair in the shape byte has at least one bit set or not. The mask generation works as follows : Take the shape byte. Shift it one bit to the right. "OR" these two together. The even bits will be set if either bit in each pair was set.

```
eg % 0 0 0 1 1 0 1 1
OR % 0 0 0 0 1 1 0 1
=  % 0 0 0 1 1 1 1 1
```

GET has the same structure, but with each pair of bytes it takes from the screen, it shifts the relevant bits into the A register. From here they are masked and stored in the save area. The shifting is done using a subroutine, whose address is given in TEMP. TEMP is set up at the start of the GET routine.

#### THE MAIN PROGRAM :

Various parts of the robot, the legs for example, go through a sequence of shapes. These sequences are stored as circularly linked lists of nodes. Each node is a pair of addresses. It contains the address of the shape for that stage in the sequence, and the address of the next node in the sequence. The last node in the sequence contains a pointer to the first one. The location LEGS contains the address of the node for the current stage. To illustrate, in the program, LEGS is a variable which keeps track of what position the legs are in. LEGS starts off pointing to node LEGSA. When it is time to put the legs on the screen, the shape with address given in LEGSA is put on the screen. Then, before the next set of legs is put on the screen, LEGS is made to point to the next node, LEGSB. Thus the shape of the next set of legs will be that pointed to by LEGSB, and so on.

If we watch the screen while PUT is doing something to it, it looks messy. To avoid seeing this, two screens are used. One is displayed while the other one is being worked on. Then the second is displayed, and the first is worked on. SLISTA and SLISTB contain relevant information for screens A and B respectively. They each give the address of the other, the start of screen memory for their own screen, and the location of that screen's background save area ( This is where the



background underneath the figure is stored before the shapes are laid on top ). SLISTP points to whichever one of these refers to the screen currently being worked on. A SYNC command is used to synchronise the screen change with the 50Hz field sync interrupt. This prevents the screen change from happening while the picture information is being sent to the T.V., and keeps the picture clean.

The shapes must be put to the screen in order of depth. The furthest shape must be put first, and the nearest last. The backgrounds from the background save areas must put back to the screen in the opposite order to which they were taken.

#### HOW TO RUN THE PROGRAM :

Allocate space for the program using CLEAR. Load the assembled program into memory, and run the program using EXEC. The screen will appear half white, half green. A robot on the left side of the screen will walk towards the right side of the screen. After it has gone a distance the program will end and control will return to BASIC.

#### HOW TO USE THE ROUTINES IN YOUR OWN PROGRAM :

##### GET, PUT & PUTC

These routines can be used to store shapes from the screen into memory, and display shapes in memory onto the screen. The routines are written ( as usual ) in PIC, but they all call INIT. Therefore these four routines must be kept together as a group. The coordinates of the shape must be stored in XCOORD and YCOORD. The size of the shape ( see above ) must be stored in XSIZE and YSIZE. An area of memory of size  $((XSIZE+7)/8) \times YSIZE$  bytes must be reserved to hold the shape. The address of this area must be in the U register. A call to GET, PUT, or PUTC will transfer the data in the desired direction.

To setup your own shapes in memory follow the example of say, SLEGA ( Note that these routines DO NOT require the first two bytes, which specify the size for SPUTC and SPUT ).

##### SGET, SPUT & SPUTC

These routines take GET, PUT & PUTC one step further. The coordinates are given by the D register, and the size is specified at the start of each shape. Although these routines are written in PIC, they call GET, PUT, PUTC and indirectly INIT. So, all these routines must be kept together. In these routines, the X register points to the area in memory. YSIZE is at X, XSIZE is at X+1, and the area for saving the shape starts at X+2. The A register gives the Y coordinate, and the B register the X coordinate. Only these 3 registers need be set up before calling the routine.

To fit all these routines together to produce animation, you

could use the main program as a guide. Create your own shapes. String them together into sequences such as that starting at ARMSA. Set up the screen characteristics SLISTA and SLISTB, perhaps adding extra addresses for background areas for other objects on the screen. Initialise the variables as the main program does before going into the loop. Then in the loop save the background areas, put the shapes onto the screen, display the screen, then fix up the other screen. Instead of the delay loop you could include sound effects using the SOUND routine of the MUSIC program. Use EXPLOD to add an explosion here or there, and the score routines to maintain a score, and you could develop some quite exciting and machine-language-fast games.

```

* FILE-DEMO
*
* GET/PUT DEMONSTRATION PROGRAM
*
* Produces an animated display of a

walking robot
0001 OE00          ORG $4000
0002 4000 203C    BRA BEGIN

*
* DEVICE ADDRESSES
*

0003 FF01          PIA1CA EQU $FF01          PIA1 control reg A
0004 FF03          PIA1CB EQU $FF03          PIA1 control reg B
0005 FF20          DAC EQU $FF20           PIA2 data A = D/A
converter
0006 FF22          DEVPIA EQU $FF22         PIA2 data B = VDG
control
0007 FF23          PIA2CB EQU $FF23         PIA2 control reg B
0008 FFC0          DEVSAM EQU $FFC0         SAM-RAM controller

0009 4002          END

*
* SCREEN VARIABLES
* in direct page
*

0010 4002          ORG 0
0011 0000          START RMB 2             start of graphics
screen
0012 0002          LENGTH RMB 2           YMAX*32
0013 0004          SAM RMB 1             value for display
mode reg
0014 0005          PIA RMB 1             value for PIA
0015 0006          LOWBIT RMB 1          mask for last 3
bits of xcoord.
0016 0007          YMAX RMB 1            number of lines in
screen
0017 0008          REORG

*
* CONSTANTS FOR SCREEN VARIABLES
*

0018 0600          GSTART EQU $600        Start of graphics
pages

* values of YMAX for modes 1,3 &4
* Note : LENGTH=32*YMAX
0019 0060          YMAX1 EQU 96
0020 00C0          YMAX3 EQU 192
0021 00C0          YMAX4 EQU 192
* values of SAM for modes 1,3 & 4
0022 0004          SAM1 EQU 4

```

```

0023 0006          SAM3 EQU 6
0024 0006          SAM4 EQU 6

          * values of PIA for modes 1,3 & 4
          * add CSET0/1 for colour set 0/1
0025 00C0          PIA1 EQU $C0
0026 00E0          PIA3 EQU $E0
0027 00F0          PIA4 EQU $F0
0028 0000          CSET0 EQU 0
0029 0008          CSET1 EQU 8

          * values of LOWBIT for modes 1,3 & 4
          * mask for last 3 bits of xcoordinate
          * also force xcoord. even if 4 colour mode
0030 00C6          LOWB1 EQU 6
0031 0006          LOWB3 EQU 6
0032 0007          LOWB4 EQU 7

0033 4002          END

0034 4002          ORG $10
          * variables for communication with GET,PUT
etc
0035 0010          YCOORD RMB 1          coordinates of
shape
0036 0011          XCOORD RMB 1          "
0037 0012          YSIZE RMB 1          size of shape
0038 0013          XSIZE RMB 1          "

          * variables used within GET,PUTetc
0039 0014          YCOUNT RMB 1        inner loop counter
0040 0015          XCOUNT RMB 1        outer loop counter
0041 0016          MASK RMB 2           bit mask
0042 0018          SHIFT RMB 1          multiplier to
shifta byte
0043 0019          TEMP RMB 2           temporary variable
0044 001B          TEMP1 RMB 2          "

          * variables used by main program
0045 001D          SPEED RMB 2          speed of walk
(1=run $FFFF=dead)
0046 001F          LEGS RMB 2           pointer to next
formof legs
0047 0021          FARARM RMB 2        " for far arm
0048 0023          NERARM RMB 2        " for near arm
0049 0025          SLISTP RMB 2        pointer to list for
next SCREEN
0050 0027          BCOORD RMB 2        coords of person
0051 0029          REORG

0052 0060          NLINES EQU YMAX1     number of lines on
screen
0053 0600          STARTA EQU GSTART    screen A start
address
0054 1200          STARTB EQU STARTA+NLINES*32 screen B
start address

```

\*\*\*\*\*

\* THE FOLLOWING LISTS OF ADDRESSES CANNOT  
\* BE MADE POSITION INDEPENDENT AND MUST BE  
\* MODIFIED IF THE PROGRAM IS SHIFTED  
\*

\* sequence of shapes for leg movement  
\* Format : shape-address,next-in-sequence

```
0055 0000      SHAPE EQU 0
0056 0002      NEXTN EQU 2

0057 4002 428E4006 LEGSA FDB SLEGA,*+2
0058 4006 429E400A      FDB SLEGB,*+2
0059 400A 42AE400E      FDB SLEGC,*+2
0060 400E 42BE4002      FDB SLEGD,LEGSA
```

\* sequence of shapes for arm movement  
\* Same format

```
0061 4012 42CE4016 ARMSA FDB SARMA,*+2
0062 4016 42D6401A      FDB SARMB,*+2
0063 401A 42DF401E      FDB SARMC,*+2
0064 401E 42E94022      FDB SARMD,*+2
0065 4022 42F94026 ARMSE FDB SARME,*+2
0066 4026 42E9402A      FDB SARMD,*+2
0067 402A 42DF402E      FDB SARMC,*+2
0068 402E 42D64012      FDB SARMB,ARMSA
```

\* data for each screen  
\* format : next screen, start of screen,  
\* background save area address

```
0069 0000      NEXT EQU 0
0070 0002      TSTART EQU 2
0071 0004      BG EQU 4

0072 4032 40380600 SLISTA FDB SLISTB,STARTA
0073 4036 4313      FDB BGA
0074 4038 40321200 SLISTB FDB SLISTA,STARTB
0075 403C 4339      FDB BGB
```

\*\*\*\*\*

```
0076 403E      BEGIN EQU *
0077 403E 3408   PSHS DP          Save direct page
register
0078 4040 8644   LDA #$44
0079 4042 1F8B   TFR A,DP          SET DIRECT PAGE

0080 4044 CC6006 LDD #YMAX1*256+LOWB1 Set screen
variables
0081 4047 9707   STA YMAX          -for mode 1
0082 4049 D706   STB LOWBIT       -
0083 404B 86C8   LDA #PIA1+CSET1 -
0084 404D 9705   STA PIA          -
0085 404F 8604   LDA #SAM1        -
0086 4051 9704   STA SAM          -
0087 4053 8E8000 LDX #$8000
0088 4056 9F1D   STX SPEED
```

```

* Draw background
0089 4058 CC0600      LDD #STARTA      White top half
0090 405B DD00        STD START        -
0091 405D CC0600      LDD #NLINES*16   -
0092 4060 DD02        STD LENGTH       -
0093 4062 CC0000      LDD #0           -
0094 4065 170205      LBSR CLEAR       -
0095 4068 CC0C00      LDD #STARTA+NLINES*16 Cyan bottom
half
0096 406B DD00        STD START        -
0097 406D CC5555      LDD #55555       -
0098 4070 1701FA      LBSR CLEAR       -

* Copy from screen A to screen B
0099 4073 108EOC00    LDY #NLINES*32   Number of byte to
transfer
0100 4077 109F02      STY LENGTH       -
0101 407A 8E0600      LDX #STARTA      Copy from screen A
0102 407D CE1200      LDU #STARTB      Copy to screen B
0103 4080              COPY1 EQU *
0104 4080 EC81        LDD ,X++         take 2 bytes
0105 4082 EDC1        STD ,U++         transfer them
0106 4084 313E        LEAY -2,Y        loop Y/2 times
0107 4086 26F8        BNE COPY1

* Initialise object
0108 4088 CC4002      LDD #LEGSA       Start legs at LEGSA
0109 408B DD1F        STD LEGS         -
0110 408D CC4012      LDD #ARMSA       Start far arm at
ARMSA
0111 4090 DD21        STD FARARM       -
0112 4092 CC4022      LDD #ARMSE       Start near arm at
ARMSE
0113 4095 DD23        STD NERARM       -
0114 4097 338C98      LEAU SLISTA,PCR  First SCREEN is A
0115 409A DF25        STU SLISTP       -

* Set background on the other SCREEN to
nothing
0116 409C EEC4        LDU NEXT,U       Set background
0117 409E AE44        LDX BG,U         -on other screen
0118 40A0 CCFFFF      LDD #-1          -to be invalid
0119 40A3 ED84        STD ,X           -Coords=-1

* Set initial coords of person
0120 40A5 CC1F00      LDD #(NLINES/2-17)*256 17 lines
above middle
0121 40A8 DD27        STD BCOORD

0122 40AA              LOOP EQU *
0123 40AA 9E1D        LDX SPEED        Delay for a while
0124 40AC              DELAY1 EQU *
0125 40AC 301F        LEAX -1,X
0126 40AE 26FC        BNE DELAY1

0127 40B0 DC27        LDD BCOORD       New xcoord=old
xcoord + 2
0128 40B2 CB02        ADDB #2          -

```

0129 40B4 DD27	STD BCOORD	-
0130 40B6 C1C8	CMPB #200	Finish if it has
gone		
0131 40B8 2453	BHS FINISH	-far enough
0132 40BA 109E25	LDY SLISTP	Pointer to list of
screen data		
0133 40BD AE22	LDX TSTART,Y	Set start of screen
0134 40BF 9F00	STX START	-
0135 40C1 AE24	LDX BG,Y	Save to background
0136 40C3 ED81	STD ,X++	-Save coordinates
0137 40C5 8D64	BSR SGET	-Get it
0138 40C7 8B04	ADDA #4	Move down 4 lines
0139 40C9 DE21	LDU FARARM	Pointer to far-arm
shape node		
0140 40CB AEC4	LDX SHAPE,U	Pointer to shape
0141 40CD EE42	LDU NEXTN,U	Pointer to next
node insequence		
0142 40CF DF21	STU FARARM	Save pointer
0143 40D1 8D4A	BSR SPUTC	Put arm here
0144 40D3 8004	SUBA #4	Move up again
0145 40D5 CB04	ADDB #4	and across to the
right		
0146 40D7 308D022C	LEAX SBODY,PCR	Put body here
0147 40DB 8D40	BSR SPUTC	-
0148 40DD 8B0A	ADDA #10	Move down 10 lines
0149 40DF C004	SUBB #4	and left again
0150 40E1 DE1F	LDU LEGS	Pointer to legs
shapenode		
0151 40E3 AEC4	LDX SHAPE,U	Pointer to shape
0152 40E5 EE42	LDU NEXTN,U	Pointer to next
node insequence		
0153 40E7 DF1F	STU LEGS	Save pointer
0154 40E9 8D32	BSR SPUTC	Put legs here
0155 40EB 8006	SUBA #6	Move up again
0156 40ED DE23	LDU NERARM	Pointer to near-arm
shape node		
0157 40EF AEC4	LDX SHAPE,U	Pointer to shape
0158 40F1 EE42	LDU NEXTN,U	Pointer to next
node insequence		
0159 40F3 DF23	STU NERARM	Save pointer
0160 40F5 8D26	BSR SPUTC	-
0161 40F7 13	SYNC	Synchronise with
50Hz interrupt		
0162 40F8 170149	LBSR SCREEN	-so the screen
clean is clean		
0163 40FB 10AEA4	LDY NEXT,Y	Now go to the
otherscreen		
0164 40FE 109F25	STY SLISTP	-
0165 4101 AE22	LDX TSTART,Y	Set screen start
0166 4103 9F00	STX START	-

0167 4105 AE24	LDX BG,Y	Restore background
onthis screen		
0168 4107 EC81	LDD ,X++	-Get coords
0169 4109 8D04	BSR SPUT	-Put background
back		
0170 410B 209D	BRA LOOP	Keep going
0171 410D 3588	FINISH PULS DP,PC	Return
	*	
	* SGET/SPUT/SPUTC ROUTINES	
	*	
	* These routines are position independent,	
	* but since they call GET, PUT and PUTC,	
	* they must be kept together with these	
routines		
	* for the BSRs to work properly.	
	* SPUT	
	*	
	* Put shape located at X+2 ontoscreen at	
coordinates		
given at X.	* given in D. The size of the object is	
	* No registers modified	
0172 410F	SPUT EQU *	
0173 410F 3446	PSHS U,D	
0174 4111 DD10	STD YCOORD	set coordinates to
D		
0175 4113 EC84	LDD ,X	get size
0176 4115 DD12	STD YSIZE	set variables
0177 4117 3302	LEAU 2,X	get start address
of shape		
0178 4119 8D1E	BSR PUT	put it
0179 411B 35C6	PULS U,D,PC	return
	* SPUTC	
	*	
	* PUTC shape located at X+2 onto screen at	
coordinates		
given at X.	* given in D. The size of the object is	
	* No registers modified	
0180 411D	SPUTC EQU *	
0181 411D 3446	PSHS U,D	
0182 411F DD10	STD YCOORD	set coordinates to
D		
0183 4121 EC84	LDD ,X	get size
0184 4123 DD12	STD YSIZE	set size variables
0185 4125 3302	LEAU 2,X	get start address
of shape		
0186 4127 8D4A	BSR PUTC	PUTC it
0187 4129 35C6	PULS U,D,PC	return



\* SGET  
 \*  
 \* Get shape at coordinates given in D.  
 \* The size of the object is given at X.  
 \* The save area starts at X+2  
 \* No registers modified

0188 412B	SGET	EQU *	
0189 412B 3446		PSHS U,D	
0190 412D DD10		STD YCOORD	set coordinates
0191 412F EC84		LDD ,X	get size
0192 4131 DD12		STD YSIZE	set size variables
0193 4133 3302		LEAU 2,X	get save area
address			
0194 4135 8D7B		BSR GET	get shape
0195 4137 35C6		PULS U,D,PC	return

\*  
 \* GET/PUT/PUTC ROUTINES  
 \*  
 \* These routines are position independent,  
 \* but since they call INIT,  
 \* they must be kept together with INIT  
 \* for the BSRs to work properly.

\* PUT  
 \*  
 \* Put shape at U, of size XSIZExYSIZE  
 \* at position (XCOORD,YCOORD) on the

screen

\* erasing the background.  
 \* No registers modified

0196 4139	PUT	EQU *	
0197 4139 3456		PSHS D,X,U	
0198 413B 1700B6		LBSR INIT	Initialise
0199 413E	PUT2	EQU *	A=unshifted mask
0200 413E D618		LDB SHIFT	Shift mask
0201 4140 3D		MUL	-
0202 4141 58		LSLB	-
0203 4142 49		ROLA	-
0204 4143 43		COMA	Save complemented
mask			
0205 4144 53		COMB	-
0206 4145 DD16		STD MASK	-
0207 4147 301F		LEAX -1,X	Move left
0208 4149 9F1B		STX TEMP1	Save position
0209 414B 9612		LDA YSIZE	Set up inner loop
counter			
0210 414D 9714		STA YCOUNT	-
0211 414F	PUT3	EQU *	
0212 414F A6C0		LDA ,U+	Get shape byte
0213 4151 D618		LDB SHIFT	Shift it
0214 4153 3D		MUL	-
0215 4154 58		LSLB	-
0216 4155 49		ROLA	-

0217	4156	DD19	STD TEMP	Save shifted byte
0218	4158	EC84	LDD ,X	Get 2 bytes from
screen				
0219	415A	9416	ANDA MASK	Mask off bits to
change				
0220	415C	D417	ANDB MASK+1	-
0221	415E	D319	ADD TEMP	ADD on shifted
shapebyte				
0222	4160	ED84	STD ,X	Put it back to the
screen				
0223	4162	308820	LEAX 32,X	Move down
0224	4165	0A14	DEC YCOUNT	Loop until YCOUNT=0
0225	4167	26E6	BNE PUT3	-
0226	4169	9E1B	LDX TEMP1	Restore position
0227	416B	86FF	LDA #\$FF	Next mask uses all
8 bits				
0228	416D	0A15	DEC XCOUNT	Loop until XCOUNT=0
0229	416F	26CD	BNE PUT2	-
0230	4171	35D6	PULS D,X,U,PC	
			* PUTC ROUTINE	
			*	
			* Put shape at U, of size XSIZExYSIZE	
			* at position (XCOORD,YCOORD) on the	
screen				
			* colour 0 (buff/green) is taken as	
transparent				
			* No registers modified	
0231	0055		EVBITS EQU %01010101	even bits
0232	4173		PUTC EQU *	
0233	4173	3456	PSHS D,X,U	
0234	4175	17007C	LBSR INIT	Initialise
0235	4178		PUTC2 EQU *	
0236	4178	301F	LEAX -1,X	Move left
0237	417A	9F1B	STX TEMP1	Save position
0238	417C	9612	LDA YSIZE	Set up inner loop
counter				
0239	417E	9714	STA YCOUNT	-
0240	4180		PUTC3 EQU *	
0241	4180	A6C0	LDA ,U+	Get shape byte.
0242	4182	D618	LDB SHIFT	Shift it.
0243	4184	3D	MUL	-
0244	4185	DD16	STD MASK	-Save byte, 1 bit
offposition.				
0245	4187	58	LSLB	-
0246	4188	49	ROLA	-
0247	4189	DD19	STD TEMP	Save shifted byte
0248	418B	9A16	ORA MASK	OR shifted byte
with itself				
0249	418D	DA17	ORB MASK+1	-one bit to the
right.				
0250	418F	8455	ANDA #EVBITS	Check the even
bits, they				
0251	4191	C455	ANDB #EVBITS	-will tell us which

bit pairs			
0252 4193 DD16	STD MASK		-are not
transparent (non-zero).			
0253 4195 58	LSLB		Expand those bits
into masks			
0254 4196 48	LSLA		-for the bit pairs
0255 4197 D316	ADDD MASK		-
0256 4199 43	COMA		We need a
complemented mask			
0257 419A 53	COMB		-
0258 419B A484	ANDA ,X		AND mask with 2
screenbytes			
0259 419D E401	ANDB 1,X		-
0260 419F D319	ADDD TEMP		ADD on shifted
shapebyte			
0261 41A1 ED84	STD ,X		Put it back to the
screen			
0262 41A3 308820	LEAX 32,X		Move down
0263 41A6 0A14	DEC YCOUNT		Loop until YCOUNT=0
0264 41A8 26D6	BNE PUTC3		-
0265 41AA 9E1B	LDX TEMP1		Restore position
0266 41AC 0A15	DEC XCOUNT		Loop until XCOUNT=0
0267 41AE 26C8	BNE PUTC2		-
0268 41B0 35D6	PULS D,X,U,PC		Return
	* GET		
	*		
	* Get shape of size XSIZExYSIZE		
	* from position (XCOORD,YCOORD) on the		
screen			
	* saving it at U.		
	* No registers modified		
0269 41B2	GET EQU *		
0270 41B2 3476	PSHS D,X,U,Y		
	* SET UP ADDRESS OF SHIFT LEFT		
	* SUBROUTINE FROM XCOORD		
0271 41B4 D611	LDB XCOORD		Set up address of
shift left			
0272 41B6 D406	ANDB LOWBIT		-subroutine. B=# of
bits to shift			
0273 41B8 58	ASLB		-Address=SHLEND-2*B
0274 41B9 50	NEGB		--
0275 41BA 318D0035	LEAY SHLEND,PCR		--
0276 41BE 31A5	LEAY B,Y		--
0277 41C0 8D32	BSR INIT		Initialise
0278 41C2	GET2 EQU *		
0279 41C2 9716	STA MASK		save mask
0280 41C4 301F	LEAX -1,X		move left
0281 41C6 9F1B	STX TEMP1		Save position
0282 41C8 9612	LDA YSIZE		Set up inner loop
counter			
0283 41CA 9714	STA YCOUNT		-
0284 41CC	GET3 EQU *		
0285 41CC EC84	LDD ,X		Get 2 bytes from

```

screen
0286 41CE ADA4      JSR ,Y           Shift it left
required amount
0287 41D0 9416      ANDA MASK        Mask off unwanted
bits
0288 41D2 A7C0      STA ,U+          Save it

0289 41D4 308820    LEAX 32,X        Move down
0290 41D7 0A14      DEC YCOUNT      Loop until YCOUNT=0
0291 41D9 26F1      BNE GET3         -
0292 41DB 86FF      LDA #$FF         Use all 8 bits in
next mask
0293 41DD 9E1B      LDX TEMP1        Restore position
0294 41DF 0A15      DEC XCOUNT      Loop until XCOUNT=0
0295 41E1 26DF      BNE GET2         -

0296 41E3 35F6      PULS D,X,U,Y,PC Return
* SUBROUTINES TO SHIFT D REG LEFT
0297 41E5 58        LSLB             shift left 7 bits
0298 41E6 49        ROLA
0299 41E7 58        LSLB             shift left 6 bits
0300 41E8 49        ROLA
0301 41E9 58        LSLB             shift left 5 bits
0302 41EA 49        ROLA
0303 41EB 58        LSLB             shift left 4 bits
0304 41EC 49        ROLA
0305 41ED 58        LSLB             shift left 3 bits
0306 41EE 49        ROLA
0307 41EF 58        LSLB             shift left 2 bits
0308 41F0 49        ROLA
0309 41F1 58        LSLB             shift left 1 bit
0310 41F2 49        ROLA
0311 41F3 39        SHLEND RTS

* INIT
*
* Initialisation routine for GET,PUT &
PUTC
* Note error recovery at INITO

0312 41F4          INIT EQU *
0313 41F4 9612      LDA YSIZE        Test size is
non-zero
0314 41F6 2728      BEQ INITO        -
0315 41F8 9613      LDA XSIZE        -
0316 41FA 2724      BEQ INITO        -
0317 41FC 8B07      ADDA #7          Calculate XSIZE'/8
0318 41FE 44        LSRA             -(see text)
0319 41FF 44        LSRA             - and save in
XCOUNT
0320 4200 44        LSRA             -
0321 4201 9715      STA XCOUNT      -
0322 4203 DC10      LDD YCOORD       Check that the
coordinates
0323 4205 9107      CMPA YMAX        -are on the screen
0324 4207 2417      BHS INITO        -

```

0325 4209 170018 address	LBSR CONVRT	Setup starting
0326 420C D718	STB SHIFT	-and SHIFT,
0327 420E 9615 start	LDA XCOUNT	-noting that we
0328 4210 3086 side	LEAX A,X	-from the right
0329 4212 86FF to use on	LDA # $\$FF$	Calculate the mask
0330 4214 D613 bytesfrom XSIZE	LDB XSIZE	the rightmost
0331 4216 D406 ones	ANDB LOWBIT	-Mask=(XSIZE mod 8)
0332 4218 2705 is a multiple o f 8	BEQ INIT2	-or 8 ones if XSIZE
0333 421A mod 8)	INIT1 EQU *	-Shift in (XSIZE
0334 421A 44	LSRA	--zeroes
0335 421B 5A	DECB	--
0336 421C 26FC	BNE INIT1	--
0337 421E 43	COMA	-Then complement
0338 421F 39	INIT2 RTS	
0339 4220	INIT0 EQU *	Error do nothing
0340 4220 3262 address	LEAS 2,S	Get rid of return
0341 4222 35D6 routine	PULS D,X,U,PC	Return FROM CALLING

\* end GET/PUT/PUTC routines  
 \* end SGET/SPUT/SPUTC routines  
 \* CONVRT  
 \*  
 \* Convert A=y-coordinate, B=x-coordinate  
 \* into X=address of byte on screen  
 \* and B has the corresponding bit set.  
 \* Position independent code  
 \* X,B modified

0342 4224	CONVRT EQU *	
0343 4224 3426	PSHS D,Y	save D & Y
0344 4226 44 bits	LSRA	shift D right 3
0345 4227 56	RORB	-to get offset from
0346 4228 44	LSRA	-start of screen
0347 4229 56	RORB	-
0348 422A 44	LSRA	-
0349 422B 56	RORB	-
0350 422C D300 byte	ADDD START	gives address of
0351 422E 1F01	TFR D,X	put in X reg
0352 4230 3506 again	PULS D	get coordinates
0353 4232 D406	ANDB LOWBIT	select last 3 bits
0354 4234 318D0004	LEAY CONTAB,PCR	get Bth byte from

```

table
0355 4238 E6A5          LDB B,Y          -
0356 423A 35A0          PULS Y,PC        restore Y and
return
0357 423C 80402010     CONTAB FCB $80,$40,$20,$10
0358 4240 08040201     FCB $08,$04,$02,$01
* end of CONVRT

* SCREEN
*
* Displays screen as specified by screen

variables
* Position independent code
* No registers modified

0359 4244              SCREEN EQU *
0360 4244 3416         PSHS D,X
0361 4246 B6FF22       LDA DEVPIA        mask PIA onto high
0362 4249 8407         ANDA #7           -5 bits of DEVPIA
0363 424B 9A05         ORA PIA           -
0364 424D B7FF22       STA DEVPIA        -
0365 4250 9604         LDA SAM           set 3 bits of
0366 4252 C603         LDB #3           -VDG register to
0367 4254 8EFFC0       LDX #DEVVSAM     -SAM
0368 4257 8D09         BSR SBITS        -
0369 4259 9600         LDA START        A=start address/256
0370 425B 44           LSRA             A=start address/128
0371 425C C607         LDB #7           set 7 bits of Page
Select Reg
0372 425E 8D02         BSR SBITS        -
0373 4260 3596         PULS D,X,PC
* SET B BITS OF SAM
0374 4262              SBITS EQU *
0375 4262 A781         STA ,X++         set B bits from X
zero                                     assume last bit is
0376 4264 44           LSRA             take last bit
0377 4265 2402         BCC SBO          branch if it was 0
0378 4267 A71F         STA -1,X        it was 1 so fix it
!
0379 4269 5A           SBO   DECB        loop B times
0380 426A 26F6         BNE SBITS
0381 426C 39           RTS
* end of SCREEN

* CLEAR
*
* Fills screen with pairs of bytes given

in D
* Note : does not stop if LENGTH is odd !
* Position independent code
* No registers modified

0382 426D              CLEAR EQU *
0383 426D 3430         PSHS X,Y
0384 426F 9E00         LDX START        pointer to start of

```

```

screen
0385 4271 109E02          LDY LENGTH          number of bytes to
clear
0386 4274 ED81          CLEARX STD ,X++          clear 2 bytes
0387 4276 313E          LEAY -2,Y              loop Y/2 times
0388 4278 26FA          BNE CLEARX
0389 427A 35B0          PULS X,Y,PC
* end of CLEAR

```

```

0390 427C          END
* SETUP
*
* Enable sound output from the DAC
* Must be called before using the DAC
* Position independent code
* No registers modified

```

```

0391 427C          SETUP EQU *
0392 427C 3406          PSHS D
0393 427E CCB435          LDD #$B435          set sound output
from DAC
0394 4281 B7FF01          STA PIA1CA          -disable 64
microsecond int.
0395 4284 F7FF03          STB PIA1CB          -and enable 50Hz
int.
0396 4287 863F          LDA #$3F            Enable DAC sound
0397 4289 B7FF23          STA PIA2CB          -
0398 428C 3586          PULS D,PC
* end of SETUP

```

```

* mask for data to store to DAC
* AND data with DAMASK befor storing to

```

```

DAC
0399 00FC          DAMASK EQU %11111100

```

```

0400 428E          END
* shapes used
* note format :
* vertical columns of bytes with the

```

```

rightmost
* column first.
* Colour code :
* - transparent ( when using PUTC )
* C cyan
* M magenta
* O orange

```

```

* Legs

```

```

0401 428E 070E          SLEGA FCB 7,14          size
0402 4290 00          FCB %00000000          - - - -
0403 4291 40          FCB %01000000          C - - -
0404 4292 40          FCB %01000000          C - - -
0405 4293 40          FCB %01000000          C - - -
0406 4294 10          FCB %00010000          - C - -

```

0407	4295	10	FCB	%00010000	- C - -
0408	4296	28	FCB	%00101000	- M M -
0409	4297	01	FCB	%00000001	- - - C
0410	4298	04	FCB	%00000100	- - C -
0411	4299	04	FCB	%00000100	- - C -
0412	429A	04	FCB	%00000100	- - C -
0413	429B	10	FCB	%00010000	- C - -
0414	429C	10	FCB	%00010000	- C - -
0415	429D	28	FCB	%00101000	- M M -
0416	429E	070E	SLEGB	FCB 7,14	size
0417	42A0	00	FCB	%00000000	- - - -
0418	42A1	00	FCB	%00000000	- - - -
0419	42A2	00	FCB	%00000000	- - - -
0420	42A3	40	FCB	%01000000	C - - -
0421	42A4	40	FCB	%01000000	C - - -
0422	42A5	40	FCB	%01000000	C - - -
0423	42A6	A0	FCB	%10100000	M M - -
0424	42A7	01	FCB	%00000001	- - - C
0425	42A8	01	FCB	%00000001	- - - C
0426	42A9	05	FCB	%00000101	- - C C
0427	42AA	04	FCB	%00000100	- - C -
0428	42AB	10	FCB	%00010000	- C - -
0429	42AC	80	FCB	%10000000	M - - -
0430	42AD	20	FCB	%00100000	- M - -
0431	42AE	070E	SLEGC	FCB 7,14	size
0432	42B0	00	FCB	%00000000	- - - -
0433	42B1	00	FCB	%00000000	- - - -
0434	42B2	40	FCB	%01000000	C - - -
0435	42B3	00	FCB	%00000000	- - - -
0436	42B4	00	FCB	%00000000	- - - -
0437	42B5	00	FCB	%00000000	- - - -
0438	42B6	80	FCB	%10000000	M - - -
0439	42B7	01	FCB	%00000001	- - - C
0440	42B8	01	FCB	%00000001	- - - C
0441	42B9	01	FCB	%00000001	- - - C
0442	42BA	05	FCB	%00000101	- - C C
0443	42BB	21	FCB	%00100001	- M - C
0444	42BC	09	FCB	%00001001	- - M C
0445	42BD	02	FCB	%00000010	- - - M
0446	42BE	070E	SLEGD	FCB 7,14	size
0447	42C0	00	FCB	%00000000	- - - -
0448	42C1	40	FCB	%01000000	C - - -
0449	42C2	10	FCB	%00010000	- C - -
0450	42C3	10	FCB	%00010000	- C - -
0451	42C4	40	FCB	%01000000	C - - -
0452	42C5	A0	FCB	%10100000	M M - -
0453	42C6	00	FCB	%00000000	- - - -
0454	42C7	01	FCB	%00000001	- - - C
0455	42C8	01	FCB	%00000001	- - - C



0456	42C9	01	FCB	%00000001	- - - C
0457	42CA	04	FCB	%00000100	- - C -
0458	42CB	04	FCB	%00000100	- - C -
0459	42CC	04	FCB	%00000100	- - C -
0460	42CD	0A	FCB	%00001010	- - M M

\* An arm

0461	42CE	0608	SARMA	FCB 6,8	size
0462	42D0	03		FCB %00000011	- - - 0
0463	42D1	0C		FCB %00001100	- - 0 -
0464	42D2	0C		FCB %00001100	- - 0 -
0465	42D3	30		FCB %00110000	- 0 - -
0466	42D4	30		FCB %00110000	- 0 - -
0467	42D5	CO		FCB %11000000	0 - - -

0468	42D6	0708	SARMB	FCB 7,8	size
0469	42D8	03		FCB %00000011	- - - 0
0470	42D9	03		FCB %00000011	- - - 0
0471	42DA	03		FCB %00000011	- - - 0
0472	42DB	0C		FCB %00001100	- - 0 -
0473	42DC	0C		FCB %00001100	- - 0 -
0474	42DD	0C		FCB %00001100	- - 0 -
0475	42DE	30		FCB %00110000	- 0 - -

0476	42DF	0808	SARMC	FCB 8,8	size
0477	42E1	03		FCB %00000011	- - - 0
0478	42E2	03		FCB %00000011	- - - 0
0479	42E3	03		FCB %00000011	- - - 0
0480	42E4	03		FCB %00000011	- - - 0
0481	42E5	03		FCB %00000011	- - - 0
0482	42E6	03		FCB %00000011	- - - 0
0483	42E7	03		FCB %00000011	- - - 0
0484	42E8	03		FCB %00000011	- - - 0

0485	42E9	070E	SARMD	FCB 7,14	size
0486	42EB	00		FCB %00000000	- - - -
0487	42EC	00		FCB %00000000	- - - -
0488	42ED	00		FCB %00000000	- - - -
0489	42EE	00		FCB %00000000	- - - -
0490	42EF	CO		FCB %11000000	0 - - -
0491	42FO	CO		FCB %11000000	0 - - -
0492	42F1	30		FCB %00110000	- 0 - -

0493	42F2	03		FCB %00000011	- - - 0
0494	42F3	03		FCB %00000011	- - - 0
0495	42F4	03		FCB %00000011	- - - 0
0496	42F5	03		FCB %00000011	- - - 0
0497	42F6	00		FCB %00000000	- - - -
0498	42F7	00		FCB %00000000	- - - -
0499	42F8	00		FCB %00000000	- - - -

0500	42F9	060E	SARME	FCB 6,14	size
0501	42FB	00		FCB %00000000	- - - -
0502	42FC	00		FCB %00000000	- - - -
0503	42FD	00		FCB %00000000	- - - -
0504	42FE	CO		FCB %11000000	0 - - -

0505	42FF	30	FCB	%00110000	-	0	-	-
0506	4300	0C	FCB	%00001100	-	-	0	-
0507	4301	03	FCB	%00000011	-	-	-	0
0508	4302	03	FCB	%00000011	-	-	-	0
0509	4303	03	FCB	%00000011	-	-	-	0
0510	4304	00	FCB	%00000000	-	-	-	-
0511	4305	00	FCB	%00000000	-	-	-	-
0512	4306	00	FCB	%00000000	-	-	-	-

\* Head and body

0513	4307	0A03	SBODY	FCB	10,3	size		
0514	4309	FC	FCB	%11111100	0	0	0	-
0515	430A	F4	FCB	%11110100	0	0	C	-
0516	430B	FC	FCB	%11111100	0	0	0	-
0517	430C	10	FCB	%00010000	-	C	-	-
0518	430D	54	FCB	%01010100	C	C	C	-
0519	430E	54	FCB	%01010100	C	C	C	-
0520	430F	54	FCB	%01010100	C	C	C	-
0521	4310	54	FCB	%01010100	C	C	C	-
0522	4311	54	FCB	%01010100	C	C	C	-
0523	4312	54	FCB	%01010100	C	C	C	-

0524 4313                   END

\* background save areas for SCREENS A and

B								
0525	4313		BGA	RMB	2	coordinates		
0526	4315	110E		FCB	17,14	size of area		
0527	4317			RMB	(14+7)/8*17	save area		
0528	4339		BGB	RMB	2	coordinates		
0529	433B	110E		FCB	17,14	size of area		
0530	433D			RMB	(14+7)/8*17	save area		

0531 435F                   END



# APPENDIX A

## COLOUR-SET TABLE

Pmode #	Color Set	Two Color Combination	Four Color Combination
4	0	Black/Green	—
	1	Black/Buf	—
3	0	—	Green/Yellow/Blue/Red
	1	—	Buf/Cyan/Magenta/Orange
2	0	Black/Green	—
	1	Black/Buf	—
1	0	—	Green/Yellow/Blue/Red
	1	—	Buf/Cyan/Magenta/Orange
0	0	Black/Green	—
	1	Black/Buf	—

# APPENDIX B

## GRAPHICS MODES

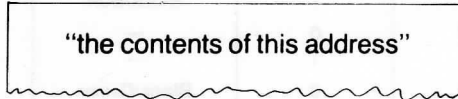
This next next section is a description of the 14 graphic modes and how to use them. This page is an explanation of the pages describing the modes. It has the same format and headings but instead of data under the heading it has an explanation of the heading.

**MODE** number and name of this mode.

**ELEMENTS** how many elements this mode has. horizontal x vertical

**MEMORY MAPPING** Where each element is stored

Addresses e.g.  
from start +0  
i.e.  
is the start +1  
address + 1



**BYTES** The number of bytes needed to hold one screen

**ADDRESS** How to calculate the address for each element. **START** is the address which the screen starts at, see page 41.

**BYTE MAPPING** What each bit in each byte of memory relates to

**COLORS** Which colors are available

**SELECT** How to set the screen to this mode

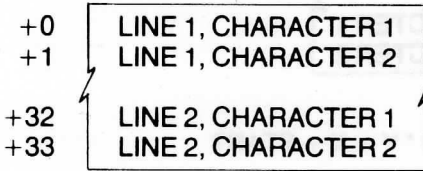
**NOTE:** That **PIA2A** is the A register in the 2nd PIA which has address \$FF22

**NOTE:** With byte mapping the elements name in a bit means that if a 1 is there the element is 'ON' and if a zero is there the element is 'OFF', an X means that that bit is not used.

## ALPHANUMERIC — NORMAL TEXT

ELEMENTS 32 x 16

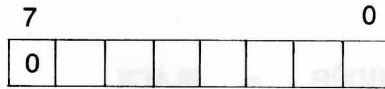
Memory mapping



BYTES = 512

ADDRESS = 32 \* Y + X + START

BYTE MAPPING



SEE APPENDIX F

COLOURS:

BORDER = BLACK

FOREGROUND COLOUR SET = 0 — GREEN

COLOUR SET = 1 — ORANGE

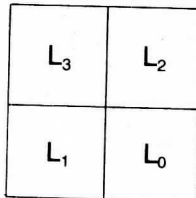
SELECT:

This is the standard screen.

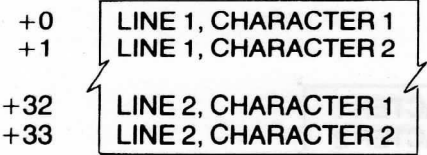
## SEMI GRAPHIC 4

ELEMENTS 64 x 32

ELEMENT FORMAT

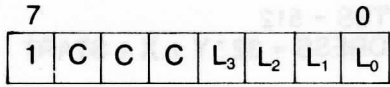


**MEMORY MAPPING**



BYTES = 512  
 ADDRESS = 32 \* Y + X + START

**BYTE MAPPING**



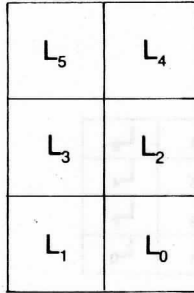
COLOURS	BORDER	=	BLACK	
	CCC	=	000	GREEN
			001	YELLOW
			010	BLUE
			011	RED
			100	BUFF
			101	CYAN
			110	MAGENTA
			111	RED

**SELECT:**  
 Standard on start up  
 Alphanumeric and this mode are together

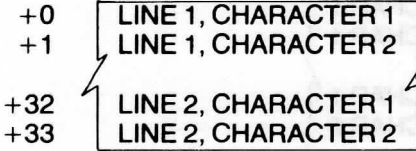
```
LDA    PIA2A
AND    #%0000 0111
STA    PIA2A
CLRA
STA    $FFC0
STA    $FFC2
STA    $FFC4
```

**SEMI GRAPHICS 6**

ELEMENTS 64 x 48  
 ELEMENT FORMAT

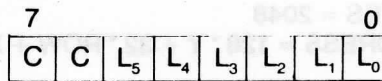


**MEMORY MAPPING**



BYTES = 512  
 ADDRESS = 32 \* Y + X + START

**BYTE MAPPING**



COLOURS BORDER = BLACK

CC	= 00	GREEN	} COLOUR SET = 0
	01	YELLOW	
	10	BLUE	
	11	RED	
	00	BUFF	} COLOUR SET = 1
	01	CYAN	
	10	MAGENTA	
	11	ORANGE	

**SELECT:**

```
LDA    PIA2A
ANDA  #$0001 0111    for colour set 0
or
ANDA  #$0001 1111    for colour set 1
CLRA
STA   $FFC4
STA   $FFC2
STA   $FFC0
```



## SEMI GRAPHICS 8

ELEMENTS 64 x 64  
ELEMENT FORMAT

A	L <sub>7</sub>	L <sub>6</sub>
B	L <sub>5</sub>	L <sub>4</sub>
C	L <sub>3</sub>	L <sub>2</sub>
D	L <sub>1</sub>	L <sub>0</sub>

### MEMORY MAPPING

+0	ROW A, LINE 1, CHAR 1
+1	ROW A, LINE 1, CHAR 1
+32	ROW B, LINE 1, CHAR 1
+33	ROW B, LINE 1, CHAR 2
+128	ROW A, LINE 2, CHAR 1
+129	ROW A, LINE 2, CHAR 2

BYTES = 2048

ADDRESS = 128 \* Y + 32 \* ROW + X + START

### BYTE MAPPING

	7						0	
A	1	C	C	C	L <sub>7</sub>	L <sub>6</sub>	X	X
B	1	C	C	C	L <sub>5</sub>	L <sub>4</sub>	X	X
C	1	C	C	C	X	X	L <sub>3</sub>	L <sub>2</sub>
D	1	C	C	C	X	X	L <sub>1</sub>	L <sub>0</sub>

COLOURS	BORDER	=	BLACK	
	CCC	=	000	GREEN
			001	YELLOW
			010	BLUE
			011	RED
			100	BUFF
			101	CYAN
			110	MAGENTA
			111	ORANGE

SELECT:

```
LDA    PIA2A
ANDA  #%0000 0111
STA    PIA2A
CLRA
STA    $FFC3
INCA
STA    $FFC3
CLRA
STA    $FFC0
```

### SEMI GRAPHICS 12

ELEMENTS 64 x 96  
ELEMENT FORMAT

A	L <sub>11</sub>	L <sub>10</sub>
B	L <sub>9</sub>	L <sub>8</sub>
C	L <sub>7</sub>	L <sub>6</sub>
D	L <sub>5</sub>	L <sub>4</sub>
E	L <sub>3</sub>	L <sub>2</sub>
F	L <sub>1</sub>	L <sub>0</sub>

### MEMORY MAPPING

+0	ROW A, LINE 1, CHAR 1
+1	ROW A, LINE 1, CHAR 2
+32	ROW B, LINE 1, CHAR 1
+33	ROW B, LINE 1, CHAR 2
+192	ROW A, LINE 2, CHAR 1
+193	ROW A, LINE 2, CHAR 2

BYTES = 3072

ADDRESS = Y \* 192 + ROW \* 32 + X + START

### BYTE MAPPING

	7							0
A	1	C	C	C	L <sub>11</sub>	L <sub>10</sub>	X	X
B	1	C	C	C	L <sub>9</sub>	L <sub>8</sub>	X	X
C	1	C	C	C	L <sub>7</sub>	L <sub>6</sub>	X	Y
D	1	C	C	C	X	X	L <sub>5</sub>	L <sub>4</sub>
E	1	C	C	C	X	X	L <sub>3</sub>	L <sub>2</sub>
F	1	C	C	C	X	X	L <sub>1</sub>	L <sub>0</sub>

COLOURS	BORDER	=	BLACK	
	CC	=	000	GREEN
			001	YELLOW
			010	BLUE
			011	RED
			100	BUFF
			101	CYAN
			110	MAGENTA
			111	ORANGE

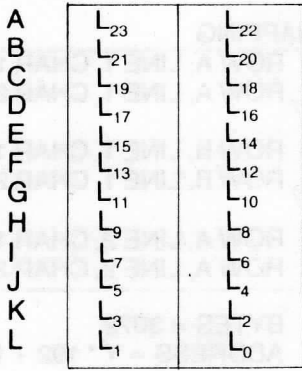
**SELECT:**

```

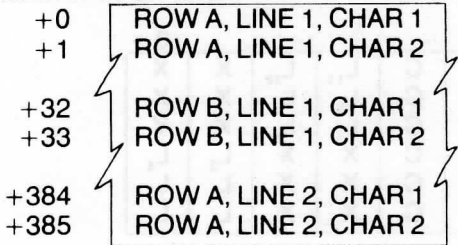
LDA    PIA2A
ANDA  #%0000 0111
STA    PIA2A
LDA    # $01
STA    $FFC5
CLRA
STA    $FFC2
STA    $FFC0
  
```

**SEMI GRAPHICS 24**

**ELEMENTS 64 x 192**  
**ELEMENT FORMAT**



**MEMORY MAPPING**



BYTES = 6144

ADDRESS =  $Y * 384 + \text{ROW} * 32 + X + \text{STARTS}$

BYTE MAPPING

	7						0	
A	1	C	C	C	L <sub>23</sub>	L <sub>22</sub>	X	X
F	1	C	C	C	L <sub>13</sub>	L <sub>12</sub>	X	X
G	1	C	C	C	X	X	L <sub>11</sub>	L <sub>10</sub>
L	1	C	C	C	X	X	L <sub>1</sub>	L <sub>0</sub>

COLOURS	BORDER	=	BLACK	
	CCC	=	000	GREEN
			001	YELLOW
			010	BLUE
			011	RED
			100	BUFF
			101	CYAN
			110	MAGENTA
			111	ORANGE

SELECT:

```
LDA PIA2A
ANDA #%0000 0111
STA PIA2A
LDA#$01
STA $FFC5
STA $FFC3
CLRA
STA $FFC0
```

GRAPHICS 64 x 64 FOUR COLOUR

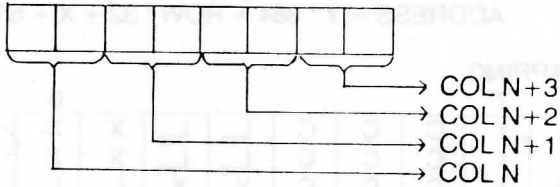
MEMORY MAPPING

+0	ROW 1, COLUMN 1—4
+1	ROW 1, COLUMN 5—8
+16	ROW 2, COLUMN 1—4
+17	ROW 2, COLUMN 5—8
+1022	ROW 64, COLUMN 57—60
+1023	ROW 64, COLUMN 61—64

BYTES = 1024

ADDRESS =  $\text{ROW} * 16 + \text{FIX}((\text{COLUMN}-1)/4) + \text{START}$

## BYTE MAPPING



COLOURS	BORDER	= GREEN	COLOR SET = 0
	CC	= 00	COLOUR SET = 1
		01	GREEN
		10	YELLOW
		11	BLUE
		00	RED
		01	BUFF
		10	CYAN
		11	MAGENTA
			ORANGE

} COLOUR SET = 0

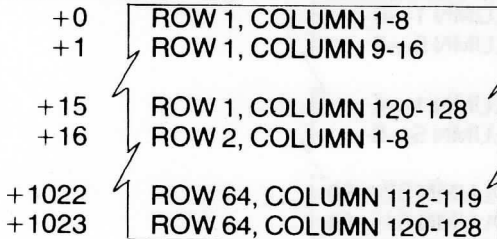
} COLOUR SET = 1

### SELECT:

```
LDA    PIA2A
ANDA  #%1000 0111    for colour set 0
or
ANDA  #%1000 1111    for colour set 1
LDA    #$01
STA    $FFC1
CLRA
STA    $FFC2
STA    $FFC4
```

## GRAPHICS 128 x 64 TWO COLOUR

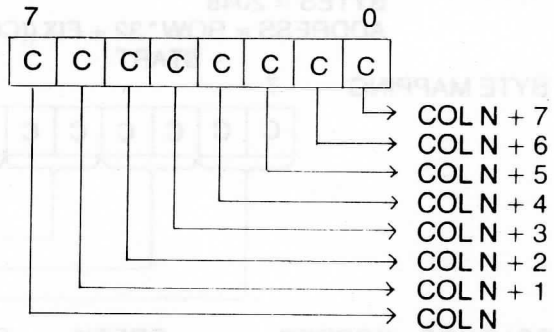
### MEMORY MAPPING



BYTES = 1024

ADDRESS = ROW\*16 + FIX((COLUMN - 1)/8) + START

## BYTE MAPPING



COLOURS	BORDER	=	GREEN	COLOUR SET = 0
			BUFF	COLOUR SET = 1
	C = 0		BLACK	COLOUR SET = 0
	1		GREEN	
	0		BLACK	COLOUR SET = 1
	1		BUFF	

## SELECT:

```

LDA    PIA2A
ANDA  #%1001 0111    for colour set 0
or
ANDA  #%1001 1111    for colour set 1
LDA    #$01
STA    $FFC1
CLRA
STA    $FFC2
STA    $FFC4
    
```

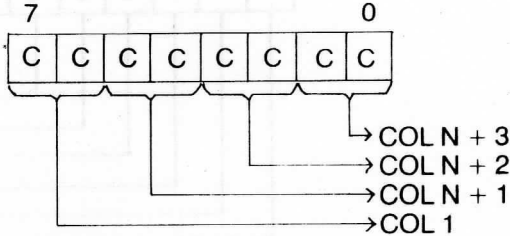
## GRAPHICS 128 x 64 FOUR COLOUR

### MEMORY MAPPING

+0	ROW 1, COL 1-4
+1	ROW 1, COL 5-8
+31	ROW 1, COL 61 - 64
+32	ROW 2, COL 1-4
+2046	ROW 64, COL 57-60
+2047	ROW 64, COL 61-64

BYTES = 2048  
 ADDRESS = ROW \* 32 + FIX ((COLUMN-1)/4)  
 + START

BYTE MAPPING



COLOURS	BORDER	=	GREEN	COLOUR SET = 0
			BUFF	COLOUR SET = 1
	CC = 00		GREEN	} COLOUR SET = 0
	01		YELLOW	
	10		BLUE	
	11		RED	
	00		BUFF	} COLOUR SET = 1
	01		CYAN	
	10		MAGENTA	
	11		ORANGE	

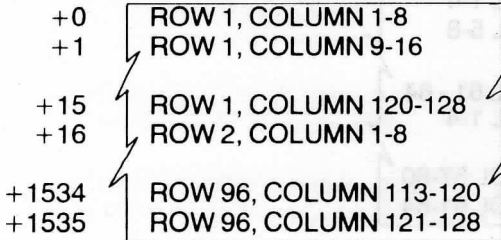
SELECT:

```

LDA   PIA2A
ANDA  #%1010 0111    for colour set 0
or
ANDA  #%1010 1111    for colour set 1
CLRA
STA   $FFC0
STA   $FFC3
INCA
STAA  $FFC4
  
```

GRAPHICS 128 x 96 TWO COLOUR PMODE 0

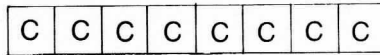
MEMORY MAPPING



BYTES = 1536

ADDRESS = ROW \* 16 + FIX ((COLUMN-1)/8) + START

BYTE MAPPING . 7



0

COLOURS BORDER = GREEN COLOUR SET = 0  
                  BUFF COLOUR SET = 1  
                  BLACK } COLOUR SET = 0  
                  GREEN }  
                  0 BLACK } COLOUR SET = 1  
                  1 BUFF }

SELECT:

LDA PIA2A  
ANDA #%1011 0111 for colour set 0  
or  
ANDA #%1011 1111 for colour set 1  
CLRA  
STA \$FFC4  
INCA  
STA \$FFC3  
STA \$FFC1

GRAPHICS 128 x 96 FOUR COLOUR PMODE 1

MEMORY MAPPING

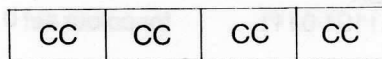
EACH BYTE HOLDS 4 COLUMNS

BYTES = 3072

ADDRESS = ROW \* 32 + FIX ((COLUMN-1)/4) + START

BYTE MAPPING

7



0

COLOURS BORDER = GREEN COLOUR SET = 0  
                  BUFF COLOUR SET = 1  
                  GREEN } COLOUR SET = 0  
                  01 YELLOW }  
                  10 BLUE }  
                  11 RED }



00	BUFF	} COLOUR SET = 1
01	CYAN	
10	MAGENTA	
11	ORANGE	

**SELECT:**

```

LDA    PIA2A
ANDA  #%1100 0111    for colour set 0
or
ANDA  #%1100 1111    for colour set 1
LDA    #$01
STA    $FFC5
CLRA
STA    $FFC2
STA    $FFC0

```

**GRAPHICS 128 x 192 TWO COLOUR PMODE 2**

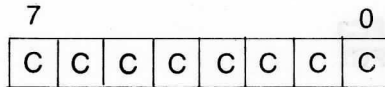
**MEMORY MAPPING**

EACH BYTE HOLDS 8 COLUMNS

BYTES = 3072

ADDRESS = ROW \* 16 + FIX ((COLUMN-1)/8) + START

**BYTE MAPPING**



COLOURS	BORDER	=	GREEN	COLOUR SET = 0
			BUFF	COLOUR SET = 1
C = 0			BLACK	} COLOUR SET = 0
1			GREEN	
0			BLACK	} COLOUR SET = 1
1			BUFF	

**SELECT:**

```

LDA    PIA2A
ANDA  #%1101 0111    for colour set 0
or
ANDA  #%1101 1111    for colour set 1
STA    PIA2A
LDA    #$01
STA    $FFC5
CLRA
STA    $FFC2
INCA
STA    $FFC1

```

**GRAPHICS 128 x 192 FOUR COLOUR PMODE 3**

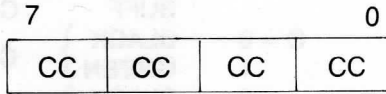
**MEMORY MAPPING**

EACH BYTE HOLDS 4 COLUMNS

BYTES = 6144

ADDRESS = ROW \* 32 + FIX ((COLUMN-1)/4) + START

**BYTE MAPPING**



COLOURS	BORDER	=	GREEN	COLOUR SET = 0
			BUFF	COLOUR SET = 1
	CC = 00		GREEN	} COLOUR SET = 0
	01		YELLOW	
	10		BLUE	
	11		RED	
	00		BUFF	} COLOUR SET = 1
	01		CYAN	
	10		MAGENTA	
	11		ORANGE	

**SELECT:**

```

LDA    PIA2A
ANDA  #%1110 0111    for colour set 0
or
ANDA  #%1110 1111    for colour set 1
STA    PIA2A
LDA    #$01
STA    $FFC5
STA    $FFC3
CLRA
STA    $FFC0
    
```

**GRAPHICS 256 x 192 TWO COLOUR PMODE 4**

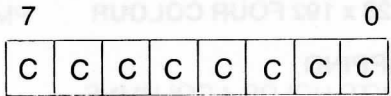
**MEMORY MAPPING**

EACH BYTE HOLDS 8 COLUMNS

BYTES 6144

ADDRESS = ROW \* 32 + FIX ((COLUMN-1)/8) + START

BYTE MAPPING



COLOURS	BORDER	=	GREEN	COLOUR SET = 0
			BUFF	COLOUR SET = 1
	C = 0		BLACK	} COLOUR SET = 0
	1		GREEN	
	0		BLACK	} COLOUR SET = 1
	1		BUFF	

SELECT:

```

LDA    PIA2A
AND A  #%1111 0111    for colour set 0
or
AND A  #%1111 1111    for colour set 1
STA    PIA2A
LDA    #$01
STA    $FFC5
STA    $FFC3
CLRA
STA    $FFC0
    
```

# APPENDIX C

## HANDY MEMORY LOCATIONS IN THE DRAGON

START ADDRESS		DESCRIPTION	END ADDRESS	
DEC	HEX		DEC	HEX
00025	0019	Address of start of BASIC program	00026	001A
00027	001B	Address of Start of variable storage also address - 1 is end of Basic program	00028	001C
00029	001D	Address of start of array storage	00030	001E
00031	001F	Address of start of free memory	00032	0020
00033	0021	Address of start of string stack	00034	0022
00035	0023	Address of BASIC upper limit	00036	0024
00039	0027	Highest available RAM address	00040	0028
00108	006C	Current column position of cursor		
00111	006F	Device number DEVNUM		
00113	0071	Warm start flag RSTFLAG &H0 Condition before cartridge program Starts created by BASIC &H12 = Do warm start &H55 = If RSTVEC points to a NOP then execute from address RSTVEC else start BASIC		
00114	0072	Warm start Vector RSTVEC	00115	0073
00116	0074	Highest physical memory address	00117	0075
00124	0076	Block type BLKTYP 0 = file header 1 = data FF = end of file		
00125	007D	Bytes in block BLKLEN		
00126	007E	Buffer address CBUFAD also program end + 1 after CLOADM	00127	007F
00128	0080	Check sum		
00129	0081	CSRERR		
00140	008C	Sound frequency		
00141	008D	Sound duration	00142	00BE
00157	009D	Transfer address after CLOADM	00158	009E
00182	00B6	Current Pmode		
00256	0100	SWI 3 vector	00258	0102
00259	0103	SWI 2 vector	00261	0105
00262	0106	SWI 1 vector	00264	0108
00265	0109	NMI vector	00267	010B
00268	010C	IRQ vector	00270	010E
00271	010F	FIRQ vector	00273	0111
00289	0121	Pointer to BASIC command Token Table	00290	0122
00290	0123	Pointer to BASIC command Jump Table	00292	0124

00294	0126	Pointer to BASIC function Token Table	00295	0127
00296	0128	Pointer to BASIC function Jump Table	00297	0129
00337	0151	<b>KEYBOARD Rollover Table</b>	00345	0159
00337	0151	Bit cleared if any bit in same column cleared		
		7 6 5 4 3 2 1 0		
00338	0152	ENTER X P H @ 8 0		
00339	0153	CLEAR Y Q I A 9 1		
00340	0154	Z R J B : 2		
00341	0155	↑ S K C ; 3		
00342	0156	↓ T L D , 4		
00343	0157	← U M E — 5		
00344	0158	→ V N F • 6		
00345	0159	spaceW O G / 7		
00346	015A	Joystick 0 — left x position		
00347	015B	Joystick 1 — left y position		
00348	015C	Joystick 2 — right x position		
00349	015D	Joystick 3 — right y position		
00466	01D2	CASSETTE file name	00473	01D9
00474	01DA	CASSETTE buffer	00731	02D8
00485	01E5	Transfer address used in CSAVEM	00486	01E6
00733	02DD	Keyboard buffer	00988	03DC
01024	0400	Text screen memory	01535	05FF
01536	0600	GRAPHICS Screen memory (in 8 pages of 1536 bytes each)	13823	35FF
03072	0C00	User RAM note can start anywhere between 03072 (0C00) and 13224 (3600) depending on graphics pages.	32767	7FFF
32816	8033	BASIC command word table	33064	8128
33108	8154	BASIC command jump table	33225	81C9
33226	81CA	BASIC function word table	33359	824F
33360	8250	BASIC function jump table	33427	8293
33449	82A9	BASIC error message table	33499	82DB
33504	82E0	BASIC interpreter	49151	BFFF
49152	C000	cartridge slot	65279	FEFF
65280	FF00	PIA (Parrallel I/O Adapter)	65521	FFF1
65522	FFF2	SWI 3 vector (contains 0100)	65523	FFF3
65524	FFF4	SWI 2 vector (contains 0103)	65525	FFF5
65526	FFF6	FIRQ vector (contains 010F)	65527	FFF7
65528	FFF8	IRQ vector (contains 010C)	65529	FFF9
65530	FFFA	SWI 1 vector (contains 0106)	65531	FFFB
65532	FFFC	NMI vector (contains 0109)	65533	FFFD
65534	FFFE	RESET vector (contains B3B4)	65535	FFFF

## **APPENDIX D**

### **HANDY ROM ROUTINES**

These following routines are included in your DRAGON's ROM and can easily be used in your machine language programs.

Each routine has a name which is used to identify it followed by an entry address in hexadecimal. Some need entry conditions such as having the A accumulator and the X index register initialized.

A brief summary of what the routine does and what the entry and exit conditions are, is also included. At the end of the section there is a list of the variables (memory locations) that are used in the routines and a brief word on what they do.

#### **INIT &HBB40**

Initialize hardware interfaces such as printer, cassette, video, memory, etc.

Shouldn't be used except for auto-start cartridge programs.

#### **SETUP &HBB88**

Sets up BASIC system variables such as keyboard debounce, cassette leader length, printer variables, etc.

#### **BLINK &HBBB5**

Decrements location 008F and when this counter reaches zero the cursor is toggled from black to green or vice-versa.

#### **TOUCH &H8E12**

Write the character in the accumulator A onto the cassette.

### BYTE-IN &HBDAD

Gets 8 bits off the cassette and puts them into the A accumulator.

### BIT-IN &HBDA5

Gets the next bit on tape and puts it into the carry bit.

### BLKIN &HB93E

Reads a block from cassette

#### CONDITIONS

ENTRY — cassette must be on and in bit synchronization (see CSRDON)

— CBUFAD(7E) contains the buffer address.

EXIT — BLKTYP(7C) contains the block type

— BLKLEN(7D) contains number of data blocks in the block (0 — 255).

— Z = 1, ACC = 0 if no errors CSRERR(81) = 0

— Z = 0, ACC = 1 if checksum error CSRERR(81) = 1

— Z = 0, ACC = 2 if memory error CSRERR(81) = 2

— Unless there was an error X points to the last byte in the buffer

— U, Y preserved, all others changed.

Interrupts are masked.

### BLKOUT &HB999

Writes a block to cassette

#### CONDITIONS

ENTRY — Tape should be up to speed

— a leader of &H55's should have been written if this is the first block to be written after motor on

— CBUFAD(7E) — buffer address

— BLKTYP(7C) — contains block type

— BLKLEN(7D) — contains number of bytes in block

EXIT — X points to last byte in buffer

— All registers modified

Interrupts are masked.

### WRTLDR &HBE6A

Turns cassette on and writes a leader.

#### CONDITIONS

ENTRY — none

EXIT — U preserved, all others modified.

### CSRDON &H8021

Turns cassette on and gets in bit sync.

#### CONDITIONS

ENTRY — none

EXIT — FIRQ and IRQ are masked.  
— U, Y preserved, all others modified.

#### CHROUT & HB54A

Outputs a character

#### CONDITIONS

ENTRY — DEVNUM(6F) set to -2 (printer) or 0 (screen).

— A character to be used.

EXIT — All registers except CC preserved.

#### JOYIN & HBD52

Sample joystick ports

#### CONDITIONS

ENTRY — none

EXIT — Y preserved, all others changed.

— POTVAL (15A) through to POTVAL + 3 (15D) contain the position of joysticks.

#### POLCAT & HBBE5

Polls keyboard for a character

#### CONDITIONS

ENTRY — none

#### EXIT

— Z = 1, A = 0 — no key pressed.

— Z = 0, A = key code — if key seen.

— B and X preserved, all others modified.

#### VARIABLES FOR ABOVE ROUTINES

BLKLEN(7D) length of cassette block

BLKTYP(7C) type of cassette block

0 = File Header

1 = Data

FF = End of File

CBUFAD(7E) cassette buffer address

CSRERR(81) cassette error type

0 = no errors

1 = checksum error

2 = memory error

DEVNUM(6F) device for CHROUT

-2 = printer

0 = screen

POTVAL(15A) 4 bytes holds current joystick position

15A = left joystick up/down

15B = left joystick left/right

15C = right joystick up/down

15D = right joystick left/right



# APPENDIX E

## ASCII CODES FOR KEYS

Key	Hex #		Decimal #	
	Unshifted	Shifted	Unshifted	Shifted
<b>BREAK</b>	03	03	03	03
<b>CLEAR</b>	0C	—	12	—
<b>ENTER</b>	0D	0D	13	13
<b>SPACEBAR</b>	20	—	32	32
!	—	21	33	—
"	—	22	34	—
#	—	23	35	—
\$	—	24	36	—
%	—	25	37	—
&	—	26	38	—
'	—	27	39	—
(	—	28	40	—
)	—	29	41	—
.	—	2A	42	—
+	—	2B	43	—
-	2D	—	45	—
=	2E	—	46	—
/	2F	—	47	—
0	30	—	48	—
1	31	—	49	—
2	32	—	50	—
3	33	—	51	—
4	34	—	52	—
5	35	—	53	—
6	36	—	54	—
7	37	—	55	—
8	38	—	56	—
9	39	—	57	—
:	3A	—	58	—
;	3B	—	59	—
<	3C	—	60	—
=	3D	—	61	—
>	3E	—	62	—
?	3F	—	63	—
@	40	13	64	19
A	61	41	97	65
B	62	42	98	66
C	63	43	99	67
D	64	44	100	68
E	65	45	101	69
F	66	46	102	70
G	67	47	103	71
H	68	48	104	72
I	69	49	105	73
J	6A	4A	106	74
K	6B	4B	107	75
L	6C	4C	108	76
M	6D	4D	109	77

Key	Hex #		Decimal #	
	Unshifted	Shifted	Unshifted	Shifted
N	6E	4E	110	78
O	6F	4F	111	79
P	70	50	112	80
Q	71	51	113	81
R	72	52	114	82
S	73	53	115	83
T	74	54	116	84
U	75	55	117	85
V	76	56	118	86
W	77	57	119	87
X	78	58	120	88
Y	79	59	121	89
Z	7A	5A	122	90
Ⓜ	5E	5F	94	95
Ⓝ	0A	5B	10	91
Ⓟ	08	15	8	21
Ⓡ	09	5D	9	93

Note: For characters A through Z, press the key combination of SHIFT 0 to utilize the upper/lowercase option. The unshifted codes will then apply.

# APPENDIX F

## CHARACTER CODES

For use with text screen.

HEX	0	10	20	30	40	50	60	70
DEC	0	16	32	48	64	80	96	112
0	0	@	P	!	0	@	P	0
1	1	A	Q	!"	1	A	Q	1
2	2	B	R	!"	2	B	R	2
3	3	C	S	!"#	3	C	S	3
4	4	D	T	!"#\$	4	D	T	4
5	5	E	U	!"#\$%	5	E	U	5
6	6	F	V	!"#\$%&	6	F	V	6
7	7	G	W	!"#\$%&'	7	G	W	7
8	8	H	X	!"#\$%&'('	8	H	X	8
9	9	I	Y	!"#\$%&'(')	9	I	Y	9
A	10	J	Z	!"#\$%&'()*	A	J	Z	A
B	11	K	[	!"#\$%&'()*+	B	K	[	B
C	12	L	\	!"#\$%&'()*+,	C	L	\	C
D	13	M	]	!"#\$%&'()*+,-	D	M	]	D
E	14	N	^	!"#\$%&'()*+,-.	E	N	^	E
F	15	O	_	!"#\$%&'()*+,-./	F	O	_	F
			←	!	0	@	P	0
			→	!"	1	A	Q	1
			↖	!"#	2	B	R	2
			↗	!"#\$	3	C	S	3
			↘	!"#\$%	4	D	T	4
			↙	!"#\$%&	5	E	U	5
			↘	!"#\$%&'	6	F	V	6
			↗	!"#\$%&'('	7	G	W	7
			↖	!"#\$%&'()*	8	H	X	8
			↘	!"#\$%&'()*+	9	I	Y	9
			↗	!"#\$%&'()*+,	A	J	Z	A
			↖	!"#\$%&'()*+,-	B	K	[	B
			↘	!"#\$%&'()*+,-.	C	L	\	C
			↗	!"#\$%&'()*+,-./	D	M	]	D
			↖	!"#\$%&'()*+,-./	E	N	^	E
			↘	!"#\$%&'()*+,-./	F	O	_	F

Note: these columns are in inverse video, i.e. green on black

# APPENDIX G

## BASE CONVERSIONS

The following table lists base conversions for all one-byte values.

DEC.	BINARY	HEX.	OCT.
0	00000000	00	000
1	00000001	01	001
2	00000010	02	002
3	00000011	03	003
4	00000100	04	004
5	00000101	05	005
6	00000110	06	006
7	00000111	07	007
8	00001000	08	010
9	00001001	09	011
10	00001010	0A	012
11	00001011	0B	013
12	00001100	0C	014
13	00001101	0D	015
14	00001110	0E	016
15	00001111	0F	017
16	00010000	10	020
17	00010001	11	021
18	00010010	12	022
19	00010011	13	023
20	00010100	14	024
21	00010101	15	025
22	00010110	16	026
23	00010111	17	027
24	00011000	18	030
25	00011001	19	031
26	00011010	1A	032
27	00011011	1B	033
28	00011100	1C	034
29	00011101	1D	035

DEC.	BINARY	HEX.	OCT.
30	00011110	1E	036
31	00011111	1F	037
32	00100000	20	040
33	00100001	21	041
34	00100010	22	042
35	00100011	23	043
36	00100100	24	044
37	00100101	25	045
38	00100110	26	046
39	00100111	27	047
40	00101000	28	050
41	00101001	29	051
42	00101010	2A	052
43	00101011	2B	053
44	00101100	2C	054
45	00101101	2D	055
46	00101110	2E	056
47	00101111	2F	057
48	00110000	30	060
49	00110001	31	061
50	00110010	32	062
51	00110011	33	063
52	00110100	34	064
53	00110101	35	065
54	00110110	36	066
55	00110111	37	067
56	00111000	38	070
57	00111001	39	071
58	00111010	3A	072
59	00111011	3B	073

DEC.	BINARY	HEX.	OCT.
60	00111100	3C	074
61	00111101	3D	075
62	00111110	3E	076
63	00111111	3F	077
64	01000000	40	100
65	01000001	41	101
66	01000010	42	102
67	01000011	43	103
68	01000100	44	104
69	01000101	45	105
70	01000110	46	1060
71	01000111	47	107
72	01001000	48	110
73	01001001	49	111
74	01001010	4A	112
75	01001011	4B	113
76	01001100	4C	114
77	01001101	4D	115
78	01001110	4E	118
79	01001111	4F	117
80	01010000	50	120
81	01010001	51	121
82	01010010	52	122
83	01010011	53	123
84	01010100	54	124
85	01010101	55	125
86	01010110	56	126
87	01010111	57	127
88	01011000	58	130
89	01011001	59	131
90	01011010	5A	132
91	01011011	5B	133
92	01011100	5C	134
93	01011101	5D	135

DEC.	BINARY	HEX.	OCT.
94	01011110	5E	136
95	01011111	5F	137
96	01100000	60	140
97	01100001	61	141
98	01100010	62	142
99	01100011	63	143
100	01100100	64	144
101	01100101	65	145
102	01100110	66	146
103	01100111	67	147
104	01101000	68	150
105	01101001	69	151
106	01101010	6A	152
107	01101011	6B	153
108	01101100	6c	154
109	01101101	6D	1550
110	01101110	6E	156
111	01101111	6F	157
112	01110000	70	160
113	01110001	71	161
114	01110010	72	162
115	01110011	73	163
116	01110100	74	164
117	01110101	75	165
118	01110110	76	166
119	01110111	77	167
120	01111000	78	170
121	01111001	79	171
122	01111010	7A	172
123	01111011	7B	173
124	01111100	7C	174
125	01111101	7D	175
126	01111110	7E	176
127	01111111	7F	177

DEC.	BINARY	HEX.	OCT.
128	10000000	80	200
129	10000001	81	201
130	10000010	82	202
131	10000011	83	203
132	10000100	84	204
133	10000101	85	205
134	10000110	86	206
135	10000111	87	207
136	10001000	88	210
137	10001001	89	211
138	10001010	8A	212
139	10001011	8B	213
140	10001100	8C	214
141	10001101	8D	215
142	10001110	8E	216
143	10001111	8F	217
144	10010000	90	220
145	10010001	91	221
146	10010010	92	222
147	10010011	93	223
148	10010100	94	224
149	10010101	95	255
150	10010110	96	226
151	10010111	97	227
152	10011000	98	230
153	10011001	99	231
154	10011010	9A	232
155	10011011	9B	233
156	10011100	9C	234
157	10011101	9D	235
158	10011110	9E	236
159	10011111	9F	237
160	10100000	A0	240
161	10100001	A1	241

DEC.	BINARY	HEX.	OCT.
162	10100010	A2	242
163	10100011	A3	243
164	10100100	A4	244
165	10100101	A5	245
166	10100110	A6	246
167	10100111	A7	247
168	10101000	A8	250
169	10101001	A9	251
170	10101010	AA	252
171	10101011	AB	253
172	10101100	AC	254
173	10101101	AD	255
174	10101110	AE	256
175	10101111	AF	257
176	10110000	B0	260
177	10110001	B1	261
178	10110010	B2	262
179	10110011	B3	263
180	10110100	B4	264
181	10110101	B5	265
182	10110110	B6	266
183	10110111	B7	267
184	10111000	B8	270
185	10111001	B9	271
186	10111010	BA	272
187	10111011	BB	273
188	10111100	BC	274
189	10111101	BD	275
190	10111110	BE	276
191	10111111	BF	277
192	11000000	C0	300
193	11000001	C1	301
194	11000010	C2	302
195	11000011	C3	303

DEC.	BINARY	HEX.	OCT.
196	11000100	C4	304
197	11000101	C5	305
198	11000110	C6	306
199	11000111	C7	307
200	11001000	C8	310
201	11001001	C9	311
202	11001010	CA	312
203	11001011	CB	313
204	11001100	CC	314
205	11001101	CD	315
206	11001110	CE	316
207	11001111	CF	317
208	11010000	D0	320
209	11010001	D1	321
210	11010010	D2	322
211	11010011	D3	323
212	11010100	D4	324
213	11010101	D5	325
214	11010110	D6	326
215	11010111	D7	327
216	11011000	D8	330
217	11011001	D9	331
218	11011010	DA	332
219	11011011	DB	333
220	11011100	DC	334
221	11011101	DD	335
222	11011110	DE	336
223	11011111	DF	337
224	11100000	E0	340
225	11100001	E1	341
226	11100010	E2	342

DEC.	BINARY	HEX.	OCT.
227	11100011	E3	343
228	11100100	E4	344
229	11100101	E5	345
230	11100110	E6	346
231	11100111	E7	347
232	11101000	E8	350
233	11101001	E9	351
234	11101010	EA	352
235	11101011	EB	353
236	11101100	EC	354
237	11101101	ED	355
238	11101110	EE	356
239	11101111	EF	357
240	11110000	F0	360
241	11110001	F1	361
242	11110010	F2	362
243	11110011	F3	363
244	11110100	F4	364
245	11110101	F5	365
246	11110110	F6	366
247	11110111	F7	367
248	11111000	F8	370
249	11111001	F9	371
250	11111010	FA	372
251	11111011	FB	373
252	11111100	FC	374
253	11111101	FD	375
254	11111110	FE	376
255	11111111	FF	377

# APPENDIX H

## 6809 INSTRUCTION SET SUMMARY

OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
00	NEG	DIRECT	6	2	1C	ANDCC	IMMED	3	2	2E	BGT	RELATIVE	3	2
03	COM		6	2	1D	SEX	INHERENT	2	1	2F	BLE	RELATIVE	3	2
04	LSR		6	2	1E	EXG		8	2	30	LEAX	INDEXED	4	2
06	ROR		6	2	1F	TFR		INHERENT	7	2	31	LEAY		4
07	ASR		6	2	20	BRA	RELATIVE	3	2	32	LEAS	4		2
08	ASL/LSL		6	2	21	BRN		3	2	33	LEAU	INDEXED	4	2
09	ROL		6	2	22	BHI		3	2	34	PSHS	INHERENT	5	2
0A	DEC		6	2	23	BLS		3	2	35	PULS		5	2
0C	INC		6	2	24	BHS/BCC		3	2	36	PSHU		5	2
0D	TST		6	2	25	BLO/BCS		3	2	37	PULU	5	2	
0E	JMP		3	2	26	BNE		3	2	39	RTS	5	1	
0F	CLR	DIRECT	6	2	27	BEQ		3	2	3A	ABX	3	1	
12	NOP	INHERENT	2	1	28	BVC		3	2	3B	RTI	6/15	1	
13	SYNC	INHERENT	2	1	29	BVS		3	2	3C	CWAI	21	2	
16	LBRA	RELATIVE	5	3	2A	BPL		3	2	3D	MUL	11	1	
17	LBSR	RELATIVE	9	3	2B	BMI	3	2	3F	SWI	19	1		
19	DAA	INHERENT	2	1	2C	BGE	3	2	40	NEGA	2	1		
1A	ORCC	IMMED	3	2	2D	BLT	RELATIVE	3	2	43	COMA	INHERENT	2	1

OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	
44	LSRA	INHERENT	2	1	5D	TSTB	INHERENT	2	1	77	ASR	EXTENDED	7	3	
46	RORA		2	1	5F	CLRB	INHERENT	2	1	78	ASL/LSL		7	3	
47	ASRA		2	1	60	NEG	INDEXED	6	2	79	ROL		7	3	
48	ASLA/LSLA		2	1	63	COM		6	2	7A	DEC		7	3	
49	ROLA		2	1	64	LSR		6	2	7C	INC		7	3	
4A	DECA		2	1	66	ROR		6	2	7D	TST		7	3	
4C	INCA		2	1	67	ASR		6	2	7E	JMP		4	3	
4D	TSTA		2	1	68	ASL/LSL		6	2	7F	CLR		EXTENDED	7	3
4F	CLRA		2	1	69	ROL		6	2	80	SUBA		IMMED	2	2
50	NEGB		2	1	6A	DEC		6	2	81	CMPA		2	2	
53	COMB		2	1	6C	INC		6	2	82	SBCA		2	2	
54	LSRB	2	1	6D	TST	6		2	83	SUBD	4	3			
56	RORB	2	1	6E	JMP	3		2	84	ANDA	2	2			
57	ASRA	2	1	6F	CLR	INDEXED	6	2	85	BITA	2	2			
58	ASLB/LSLB	2	1	70	NEG	EXTENDED	7	3	86	LDA	2	2			
59	ROLB	2	1	73	COM		7	3	88	EORA	2	2			
5A	DECB	2	1	74	LSR		7	3	89	ADCA	2	2			
5C	INCB	INHERENT	2	1	76		ROR	EXTENDED	7	3	8A	ORA	IMMED	2	2



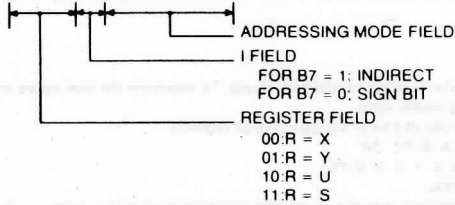
OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
8B	ADDA	IMMED	2	2	9E	LDX	DIRECT	5	2	B0	SUBA	EXTENDED	5	3
8C	CMPX	IMMED	4	3	9F	STX	DIRECT	5	2	B1	CMPA		5	3
8D	BSR	RELATIVE	7	2	A0	SUBA	INDEXED	4	2	B2	SBCA		5	3
8E	LDX	IMMED	3	3	A1	CMPA		4	2	B3	SUBD		7	3
90	SUBA	DIRECT	4	2	A2	SBCA		4	2	B4	ANDA		5	3
91	CMPA		4	2	A3	SUBD		6	2	B5	BITA		5	3
92	SBCA		4	2	A4	ANDA		4	2	B6	LDA		5	3
93	SUBD		6	2	A5	BITA		4	2	B7	STA		5	3
94	ANDA		4	2	A6	LDA		4	2	B8	EORA		5	3
95	BITA		4	2	A7	STA		4	2	B9	ADCA		5	3
96	LDA		4	2	A8	EORA		4	2	BA	ORA		5	3
97	STA		4	2	A9	ADCA		4	2	BB	ADDA		5	3
98	EORA		4	2	AA	ORA		4	2	BC	CMPX		7	3
99	ADCA		4	2	AB	ADDA		4	2	BD	JSR		8	3
9A	ORA		4	2	AC	CMPX		6	2	BE	LDX		6	3
9B	ADDA		4	2	AD	JSR		7	2	BF	STX	EXTENDED	6	3
9C	CMPX		6	2	AE	LDX		5	2	C0	SUBB	IMMED	2	2
9D	JSR	DIRECT	7	2	AF	STX	INDEXED	5	2	C1	CMPB	IMMED	2	2

OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
C2	SBCB	IMMED	2	2	D7	STB	DIRECT	4	2	E9	ADCB	INDEXED	4	2
C3	ADDD		4	3	D8	EORB		4	2	EA	ORB		4	2
C4	ANDB		2	2	D9	ADCB		4	2	EB	ADDB		4	2
C5	BITB		2	2	DA	ORB		4	2	EC	LDD		5	2
C6	LDB		2	2	DB	ADDB		4	2	ED	STD		5	2
C8	EORB		2	2	DC	LDD		5	2	EE	LDU		5	2
C9	ADCB		2	2	DD	STD		5	2	EF	STU	INDEXED	5	2
CA	ORB		2	2	DE	LDU		5	2	F0	SUBB	EXTENDED	5	3
CB	ADDB		2	2	DF	STU	DIRECT	5	2	F1	CMPB		5	3
CC	LDD		3	3	E0	SUBB	INDEXED	4	2	F2	SBCB		5	3
CE	LDU	IMMED	3	3	E1	CMPB		4	2	F3	ADDD		7	3
D0	SUBB	DIRECT	4	2	E2	SBCB		4	2	F4	ANDB		5	3
D1	CMPB		4	2	E3	ADDD		6	2	F5	BITB		5	3
D2	SBCB		4	2	E4	ANDB		4	2	F6	LDB		5	3
D3	ADDD		6	2	E5	BITB		4	2	F7	STB		5	3
D4	ANDB		4	2	E6	LDB		4	2	F8	EORB		5	3
D5	BITB		4	2	E7	STB		4	2	F9	ADCB		5	3
D6	LDB	DIRECT	4	2	E8	EORB	INDEXED	4	2	FA	ORB	EXTENDED	5	3

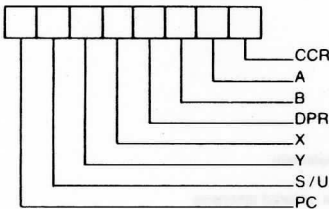
OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#	OP	MNEM	MODE	~	#
FB	ADDB	EXTENDED	5	3	102E	LBGT	RELATIVE	5(6)	4	10CE	LDS	IMMED	4	4
FC	LDD		6	3	102F	LBLE	RELATIVE	5(6)	4	10DE	LDS	DIRECT	6	3
FD	STD		6	3	103F	SWI/2	INHERENT	20	2	10DF	STS	DIRECT	6	3
FE	LDU		6	3	1083	CMPD	IMMED	5	4	10EE	LDS	INDEXED	6	3
FF	STU	EXTENDED	6	3	108C	CMPY		5	4	10EF	STS	INDEXED	6	3
1021	LBRN	RELATIVE	5	4	108E	LDY	IMMED	4	4	10FE	LDS	EXTENDED	7	4
1022	LBHI		5(6)	4	1093	CMPD	DIRECT	7	3	10FF	STS	EXTENDED	7	4
1023	LBLS		5(6)	4	109C	CMPY		7	3	113F	SWI/3	INHERENT	20	2
1024	LBHS/LBCC		5(6)	4	109E	LDY		6	3	1183	CMPU	IMMED	5	4
1025	LBCS/LBLO		5(6)	4	109F	STY	DIRECT	6	3	118C	CMPB	IMMED	5	4
1026	LBNE		5(6)	4	10A3	CMPD	INDEXED	7	3	1193	CMPU	DIRECT	7	3
1027	LBEQ		5(6)	4	10AC	CMPY		7	3	119C	CMPB	DIRECT	7	3
1028	LBVC		5(6)	4	10AE	LDY		6	3	11A3	CMPU	INDEXED	7	3
1029	LBVS		5(6)	4	10AF	STY	INDEXED	6	3	11AC	CMPB	INDEXED	7	3
102A	LBPL		5(6)	4	10B3	CMPD	EXTENDED	8	4	11B3	CMPU	EXTENDED	8	4
102B	LBMI		5(6)	4	10BC	CMPY		8	4	11BC	CMPB	EXTENDED	8	4
102C	LBGE		5(6)	4	10BE	LDY		7	4					
102D	LBLT	RELATIVE	5(6)	4	10BF	STY	EXTENDED	7	4					

## INDEXED ADDRESSING POST BYTE REGISTER BIT ASSIGNMENTS

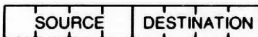
POST-BYTE REGISTER BIT								INDEXED ADDRESSING MODE
7	6	5	4	3	2	1	0	EA = ,R ± 4 BIT OFFSET
0	X	X	X	X	X	X	X	,R+
1	X	X	0	0	0	0	0	,R++
1	X	X	0	0	0	1	0	,-R
1	X	X	X	0	0	1	1	,--R
1	X	X	X	0	1	0	0	EA = ,R ± 0 OFFSET
1	X	X	X	0	1	0	1	EA = ,R ± ACCB OFFSET
1	X	X	X	0	1	1	0	EA = ,R ± ACCA OFFSET
1	X	X	X	1	0	0	0	EA = ,R ± 7 BIT OFFSET
1	X	X	X	1	0	0	1	EA = ,R ± 15 BIT OFFSET
1	X	X	X	1	0	1	1	EA = ,R ± D OFFSET)
1	X	X	X	1	1	0	0	EA = ,PC ± 7 BIT OFFSET
1	X	X	X	1	1	0	1	EA = ,PC ± 15 BIT OFFSET
1	X	X	1	1	1	1	1	EA = ,ADDRESS



### PUSH/PULL POST BYTE



### TRANSFER/EXCHANGE POST BYTE



### REGISTER FIELD

0000 = D (A,B)	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = U	1011 = DPR
0100 = S	
0101 = PC	

### 6809 STACKING ORDER

#### PULL ORDER

↓  
CC  
A  
B  
DP  
X Hi  
X Lo  
Y Hi  
Y Lo  
U/S Hi  
U/S Lo  
PC Hi  
PC Lo

#### 6809 VECTORS

FFFE Restart  
FFFC NMI  
FFFA SWI  
FFF8 IRQ  
FFF6 FIRQ  
FFF4 SWI2  
FFF2 SWI3  
FFF0 Reserved

#### ↑ PUSH ORDER

INCREASING  
MEMORY



## INDEXED ADDRESSING MODES

TYPE	FORMS	NON INDIRECT			INDIRECT		
		Assembler Form	Post-Byte OP Code	+ ~ #	Assembler Form	Post-Byte OP Code	+ ~ #
CONSTANT OFFSET FROM R	NO OFFSET	. R	1RR00100	0 0	[. R]	1RR10100	3 0
	5 BIT OFFSET	n, R	0RRnnnnn	1 0	defaults to 8-bit		
	8 BIT OFFSET	n, R	1RR01000	1 1	[n, R]	1RR11000	4 1
	16 BIT OFFSET	n, R	1RR01001	4 2	[n, R]	1RR11001	7 2
ACCUMULATOR OFFSET FROM R	A—REGISTER OFFSET	A, R	1RR00110	1 0	[A, R]	1RR10110	4 0
	B—REGISTER OFFSET	B, R	1RR00101	1 0	[B, R]	1RR10101	4 0
	D—REGISTER OFFSET	D, R	1RR01011	4 0	[D, R]	1RR11011	7 0
AUTO INCREMENT/DECREMENT R	INCREMENT BY 1	. R+	1RR00000	2 0	not allowed		
	INCREMENT BY 2	. R++	1RR00001	3 0	[. R++]	1RR10001	6 0
	DECREMENT BY 1	. -R	1RR00010	2 0	not allowed		
	DECREMENT BY 2	. --R	1RR00011	3 0	[. --R]	1RR10011	6 0
CONSTANT OFFSET FROM PC	8 BIT OFFSET	n, PCR	1XX01100	1 1	[n, PCR]	1XX11100	4 1
	16 BIT OFFSET	n, PCR	1XX01101	5 2	[n, PCR]	1XX11101	8 2
EXTENDED INDIRECT	16 BIT ADDRESS	—	—	--	[n]	10011111	5 2

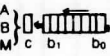
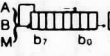
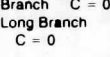
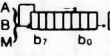
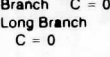
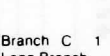
R = X, Y, U, or S  
X = DONT CARE

### NOTES:

1. Given in the table are the base cycles and byte counts. To determine the total cycles and byte counts add the values from the 6809 indexing modes table.
2. R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.  
The 8 bit registers are: A, B, CC, DP  
The 16 bit registers are: X, Y, U, S, D, PC
3. EA is the effective address.
4. The PSH and PUL instructions require 5 cycles plus 1 cycle for each byte pushed or pulled
5. 5(6) means: 5 cycles if branch not taken, 6 cycles if taken.
6. SW1 sets I&F bits. SW12 and SW13 do not affect I&F
7. Conditions Codes set as a direct result of the instruction.
8. Value of half-carry flag is undefined
9. Special Case—Carry set if b7 is SET.

### LEGEND:

OP Operation Code (Hexadecimal);	Z Zero (byte)
~ Number of MPU Cycles;	∨ Overflow, 2's complement
# Number of Program Bytes;	C Carry from bit 7
+ Arithmetic Plus;	‡ Test and set if true, cleared otherwise
- Arithmetic Minus;	• Not Affected
· Multiply	CC Condition Code Register
M Complement of M;	: Concatenation
→ Transfer Into;	v Logical or
H Half-carry from bit 3;	∧ Logical and
N Negative (sign bit)	⊘ Logical Exclusive or

INSTRUCTION/ FORMS	6809 ADDRESSING MODES												DESCRIPTION	H	N	Z	V	C							
	INHERENT			DIRECT			EXTENDED			IMMEDIATE									INDEXED <sup>1</sup>			RELATIVE			
	OP	~	#	OP	~	#	OP	~	#	OP	~	#							OP	~	#	OP	~	#	
ABX	3A	3	1																B + X → X (UNSIGNED)	*	*	*	*	*	*
ADC	ADCA ADCB			99 D9	4 2	B9 F9	5 3	89 C9	2 2	2 2	A9 E9	4+ 4+	2+ 2+						A + M + C → A B + M + C → B	↑	↑	↑	↑	↑	↑
ADD	ADDA ADDB ADDD			9B DB D3	4 2 2	BB FB F3	5 3 7	BB CB C3	2 2 3	2 2 4	AB EB E3	4+ 4+ 6+	2+ 2+ 2+						A + M → A B + M → B D + M: M + 1 → D	↑	↑	↑	↑	↑	↑
AND	ANDA ANDB ANDCC			94 D4	4 2	B4 F4	5 3	84 C4 1C	2 2 3	2 2 2	A4 E4 1C	4+ 4+ 3	2+ 2+ 2						A M → A B M → B CC IMM → CC	*	↑	↑	0	*	↑
ASL	ASLA ASLB ASL	48 58	2 2	1 1															A)  0 B)  0 M)  0	8	↑	↑	↑	↑	↑
ASR	ASRA ASR ASR	47 57	2 2	1 1	08	6	2 78	7 3				68	6+	2+					A)  0 B)  0 M)  0	8	↑	↑	↑	↑	↑
BCC	BCC LBCC														24 10 24	3 5(6) 4	2 4	Branch C = 0 Long Branch C = 0	*	*	*	*	*	*	
BCS	BCS LBCS														25 10 25	3 5(6) 4	2 4	Branch C 1 Long Branch C 1	*	*	*	*	*	*	
BEQ	BEQ LBEQ														27 10 27	3 5(6) 4	2 4	Branch Z = 0 Long Branch Z = 0	*	*	*	*	*	*	
BGE	BGE LBGE														2C 10 2C	3 5(6) 4	2 4	Branch ≥ Zero Long Branch ≥ Zero	*	*	*	*	*	*	
BGT	BGT LBGT														2E 10 2E	3 5(6) 4	2 4	Branch > Zero Long Branch > Zero	*	*	*	*	*	*	
BHI	BHI LBHI														22 10 22	3 5(6) 4	2 4	Branch Higher Long Branch Higher	*	*	*	*	*	*	
BHS	BHS LBHS														24 10 24	3 5(6) 4	2 4	Branch Higher or Same Long Branch Higher or Same	*	*	*	*	*	*	
BIT	BITA BITB				95 D5	4 4	2 B5 2 F5	5 3 5 3	85 C5	2 2 2 2	A5 E5	4+ 4+	2+ 2+						Bit Test A (M ∧ A) Bit Test B (M ∧ B)	*	↑	↑	0	*	↑
BLE	BLE LBLE														2F 10 2F	3 5(6) 4	2 4	Branch ≤ Zero Long Branch ≤ Zero	*	*	*	*	*	*	
BLO	BLO LBLO														25 10 25	3 5(6) 4	2 4	Branch Lower Long Branch Lower	*	*	*	*	*	*	
BLS	BLS LBLS														23 10 23	3 5(6) 4	2 4	Branch Lower or Same Long Branch Lower or Same	*	*	*	*	*	*	
BLT	BLT LBLT														2D 10 2D	3 5(6) 4	2 4	Branch < Zero Long Branch < Zero	*	*	*	*	*	*	
BMI	BMI LBMI														2B 10 2B	3 5(6) 4	2 4	Branch Minus Long Branch Minus	*	*	*	*	*	*	
BNE	BNE LBNE														26 10 26	3 5(6) 4	2 4	Branch Z ≠ 0 Long Branch Z ≠ 0	*	*	*	*	*	*	
BPL	BPL LBPL														2A 10 2A	3 5(6) 4	2 4	Branch Plus Long Branch Plus	*	*	*	*	*	*	

INSTRUCTION/ FORMS	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED			RELATIVE		DESCRIPTION	5	3	2	1	D
	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~						
BRA LBRA															20	3	2	Branch Always	*	*	*	*	*
															16	5	3	Long Branch Always	*	*	*	*	*
	BRN LBRN															21	3	2	Branch Never	*	*	*	*
															10	5	4	Long Branch Never	*	*	*	*	*
															21								
BSR LBSR															8D	7	2	Branch to Subroutine	*	*	*	*	*
															17	9	3	Long Branch to Subroutine	*	*	*	*	*
BVC LBVC															28	3	2	Branch V = 0	*	*	*	*	*
														10	5(6)	4	4	Long Branch V 0	*	*	*	*	*
														28									
BVS LBVS															29	3	2	Branch V = 1	*	*	*	*	*
														10	5(6)	4	4	Long Branch V = 1	*	*	*	*	*
														29									
CLR CLRA CLRB CLR	4F	2	1															0 → A	*	0	1	0	0
	5F	2	1															0 → B	*	0	1	0	0
				0F	6	2	7F	7	3				6F	6	2			0 → M	*	0	1	0	0
CMP CMPA CMPB CMPD CMPS CMPU CMPX CMPY				91	4	2	B1	5	3	B1	2	2	A1	4	2	2		Compare M from A	B	↑	↑	↑	↑
				D1	4	2	F1	5	3	C1	2	2	E1	4	2	2		Compare M from B	B	↑	↑	↑	↑
				10	7	3	10	8	4	10	5	4	10	7	3	3		Compare M from B	B	↑	↑	↑	↑
				93			B3			B3			A3					Compare M + 1	B	↑	↑	↑	↑
				11	7	3	11	8	4	11	5	4	11	7	3	3		from D					
				9C			BC			BC			AC					Compare M + 1	B	↑	↑	↑	↑
				11	7	3	11	8	4	11	5	4	11	7	3	3		from S					
				93			B3			B3			A3					Compare M + 1	B	↑	↑	↑	↑
				9C	6	2	BC	7	3	BC	4	3	AC	6	2	2		from U					
				10	7	3	10	8	4	10	5	4	10	7	3	3		Compare M + 1	B	↑	↑	↑	↑
COM COMA COMB COM	43	2	1															A → A	*	↑	↑	0	1
	53	2	1															B → B	*	↑	↑	0	1
				03	6	2	73	7	3				63	6	2	2		M → M	*	↑	↑	0	1
CWAI	3C	20	2														CC IMM -CC					1	
DAA	19	2	1															Wait for Interrupt					
DEC DECA DECB DEC	4A	2	1															Decimal Adjust A	*	↑	↑	0	↑
	5A	2	1															A - 1 → A	*	↑	↑	1	*
				0A	6	2	7A	7	3				6A	6	2	2		B - 1 → B	*	↑	↑	1	*
EOR EORA EORB				98	4	2	B8	5	3	B8	2	2	A8	4	2	2		M - 1 → M	*	↑	↑	1	*
				D8	4	2	F8	5	3	C8	2	2	E8	4	2	2		A v M - A	*	↑	↑	0	*
																		B v M - B	*	↑	↑	0	*
EXG R1, R2	1E	7	2															R1 ↔ R2	*	*	*	*	*
INC INCA INCB INC	4C	2	1															A + 1 → A	*	↑	↑	1	*
	5C	2	1															B + 1 → B	*	↑	↑	1	*
				0C	6	2	7C	7	3				6C	6	2	2		M + 1 → M	*	↑	↑	1	*
JMP	0E	3	2	7E	4	3							6E	3	2	2		EA <sup>2</sup> → PC	*	*	*	*	*
JSR	9D	7	2	BD	8	3							AD	7	2	2		Jump to Subroutine	*	*	*	*	*
LD LDA LDB LDD LDS LDS LDS LDU LDX LDY				96	4	2	B6	5	3	B6	2	2	A6	4	2	2		M - A	*	↑	↑	0	*
				D6	4	2	F6	5	3	C6	2	2	E6	4	2	2		M - B	*	↑	↑	0	*
				D0	5	2	FC	6	3	CC	3	3	EC	5	2	2		M M - 1 - D	*	↑	↑	0	*
				10	6	3	10	7	4	10	4	4	10	6	3	3		M M - 1 - S	*	↑	↑	0	*
				DE			FE			CE			EE										
				DE	5	2	FE	6	3	CE	3	3	EE	5	2	2		M M - 1 - U	*	↑	↑	0	*
				9E	5	2	BE	6	3	BE	3	3	AE	5	2	2		M M - 1 - X	*	↑	↑	0	*
				10	6	3	10	7	4	10	4	4	10	6	3	3		M M - 1 - Y	*	↑	↑	0	*
				9E			BE			BE			AE										
	LEA LEAU LEAX LEAY															32	4	2	EA <sup>2</sup> → S	*	*	*	*
															33	4	2	EA <sup>2</sup> → U	*	*	*	*	*
															30	4	2	EA <sup>2</sup> → X	*	*	*	*	*
															31	4	2	EA <sup>2</sup> → Y	*	*	*	*	*

INSTRUCTION/ FOHMS	INHERENT			DIRECT			EXTENDED			IMMEDIATE			INDEXED <sup>1</sup>			RELATIVE			DESCRIPTION	F	N	Z	V	C	
	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#							
LSL	LSLA	48	2	1															A		•	↑	↑	↑	↑
	LSLB	58	2	1															B		•	↑	↑	↑	↑
	LSL				08	6	2	78	7	3						68	6+	2+	M		•	↑	↑	↑	↑
LSR	LSRA	44	2	1															A		•	0	↑	↑	↑
	LSRB	54	2	1															B		•	0	↑	↑	↑
	LSR				04	6	2	74	7	3						64	6+	2+	M		•	0	↑	↑	↑
MUL		3D	11	1															A × B → D (Unsigned)	•	•	↑	•	9	
NEG	NEGA	40	2	1															A		8	↑	↑	↑	↑
	NEGB	50	2	1															B		8	↑	↑	↑	↑
	NEG				00	6	2	70	7	3						60	6+	2+	M		8	↑	↑	↑	↑
NOP		12	2	1															No Operation	•	•	•	•	•	
OR	ORA				9A	4	2	BA	5	3	BA	2	2	AA	4+	2+			A : M → A	•	↑	↑	0	•	
	ORB				DA	4	2	FA	5	3	CA	2	2	EA	4+	2+			B : M → B	•	↑	↑	0	•	
	ORCC										1A	3	2						CC ∨ IMM → CC	•	↑	↑	0	•	
PSH	PSHS	34	5+	2															Push Registers on S Stack	•	•	•	•	•	
	PSHU	36	5+	2															Push Registers on U Stack	•	•	•	•	•	
PUL	PULS	35	5+	2															Pull Registers from S Stack	•	•	•	•	•	
	PULU	37	5+	2															Pull Registers from U Stack	•	•	•	•	•	
ROL	ROLA	49	2	1															A		•	↑	↑	↑	↑
	ROLB	59	2	1															B		•	↑	↑	↑	↑
	ROL				09	6	2	79	7	3						69	6+	2+	M		•	↑	↑	↑	↑
ROR	RORA	46	2	1															A		•	↑	↑	↑	↑
	RORB	56	2	1															B		•	↑	↑	↑	↑
	ROR				06	6	2	76	7	3						66	6+	2+	M		•	↑	↑	↑	↑
RTI		3B	6/15	1															Return From Interrupt	•	•	•	•	7	
RTS		39	5	1															Return From Subroutine	•	•	•	•	•	
SBC	SBCA				92	4	2	B2	5	3	B2	2	2	A2	4+	2+			A - M - C → A	8	↑	↑	↑	↑	
	SBCB				D2	4	2	F2	5	3	C2	2	2	E2	4+	2+			B - M - C → B	8	↑	↑	↑	↑	
SEX		1D	2	1															Sign Extend B into A	•	↑	↑	0	•	
ST	STA				97	4	2	B7	5	3				A7	4+	2+			A → M	•	↑	↑	0	•	
	STB				D7	4	2	F7	5	3				E7	4+	2+			B → M	•	↑	↑	0	•	
	STD				DD	5	2	FD	6	3				ED	5+	2+			D → M: M + 1	•	↑	↑	0	•	
	STS				10	6	3	10	7	4				10	6+	3+			S → M: M + 1	•	↑	↑	0	•	
					DF			FF						EF							•	•	•	•	
	STU				DF	5	2	FF	6	3				EF	5+	2+			U → M: M + 1	•	↑	↑	0	•	
	STX				9F	5	2	BF	6	3				AF	5+	2+			X → M: M + 1	•	↑	↑	0	•	
	STY				10	6	3	10	7	4				10					Y → M: M + 1	•	↑	↑	0	•	
					9F			BF						AF	6+	3+					•	•	•	•	
SUB	SUBA				90	4	2	B0	5	3	B0	2	2	A0	4+	2+			A - M - A	8	↑	↑	↑	↑	
	SUBB				D0	4	2	F0	5	3	C0	2	2	E0	4+	2+			B - M - B	8	↑	↑	↑	↑	
	SUBD				93	6	2	B3	7	3	B3	4	3	A3	6+	2+			D - M: M + 1 - D	•	↑	↑	↑	↑	
SWI	SWI*	3F	19	1															Software Interrupt 1	•	•	•	•	•	
	SWI2*	10	20	2															Software Interrupt 2	•	•	•	•	•	
		3F																		•	•	•	•	•	
	SWI3*	11	20	2															Software Interrupt 3	•	•	•	•	•	
	3F																			•	•	•	•	•	
SYNC		13	≥2	1															Synchronize to Interrupt	•	•	•	•	•	
TFR	R1, R2	1F	7	2															R1 → R2*	•	•	•	•	•	
TST	TSTA	4D	2	1															Test A	•	↑	↑	0	•	
	TSTB	5D	2	1															Test B	•	↑	↑	0	•	
	TST				0D	6	2	7D	7	3					6D	6+	2+		Test M	•	↑	↑	0	•	

# INDEX

## A

accumulator . . . . . 24, 28-9, 30, 33, 34, 40, 48-9, 53  
address . . . . . 12, 21-2, 27, 32-3, 36, 56, 151, 167  
addressing mode . . . . . 26, 40  
algorithm . . . . . 178  
amplitude . . . . . 179, 191-2  
arcade game . . . . . 15  
attack . . . . . 192

## B

base . . . . . 5, 10  
basic . . . . . 15  
basic interpreter . . . . . 4, 22  
B.C.D. . . . . 11, 169  
binary . . . . . 5, 10, 9, 12, 14, 40  
binary digit . . . . . 6, 145  
bit . . . . . 6-7, 10  
branch . . . . . 45, 29  
byte . . . . . 7-8, 12, 21, 29, 37-8, 170

## C

carry . . . . . 43-4, 133-4  
character string . . . . . 151  
chip . . . . . 1, 4, 20, 156, 168  
circuit . . . . . 40  
circuit board . . . . . 20  
CLOADM . . . . . 151  
code . . . . . 18  
condition code . . . . . 59-126  
constant . . . . . 18, 128  
comment . . . . . 152  
cycles . . . . . 59-126, 191

## D

data . . . . . 23, 159  
debug . . . . . 15  
decay . . . . . 192  
digits . . . . . 6  
direct page . . . . . 160, 180  
disks . . . . . 4  
documentation . . . . . 157

## E

envelope . . . . . 191  
exponent . . . . . 9-10  
exponential . . . . . 8, 10

## F

flags . . . . . 24, 42-3, 45, 48-9  
floating point . . . . . 16  
fractions . . . . . 7  
frequency . . . . . 192  
function . . . . . 59-126

## G

graphics screen . . . . . 22, 167

## H

hexadecimal . . . . . 7, 9-10, 12, 128, 145, 152  
high byte . . . . . 24

## I

I/O . . . . . 3, 23, 156  
indices . . . . . 28  
instruction . . . . . 14-5, 17, 18, 24, 29, 32-3, 37-8, 48,  
59-127, 151-2, 170  
integer . . . . . 178  
interrupts . . . . . 45, 157, 202

## J

jargon . . . . . 1  
jumps . . . . . 143, 149

## K

"K" . . . . . 21  
keyboard . . . . . 4, 162  
keys . . . . . 162

## L

label . . . . . 18, 49  
least significant . . . . . 44, 178  
linear congruence . . . . . 178  
logical function . . . . . 24  
logical operations . . . . . 39, 139  
loop counter . . . . . 46, 48-9  
loop variable . . . . . 47  
low byte . . . . . 24

## M

mantissa . . . . . 9-10  
mask . . . . . 200  
masochistic . . . . . 37  
memory . . . . . 3-4, 16, 21-2, 28, 35, 37, 53, 127,  
157, 169  
micro . . . . . 34  
mnemonics . . . . . 17-8, 35, 59-126, 152  
mode . . . . . 25, 29-30, 32-3, 49, 57  
modules . . . . . 149  
monitor . . . . . 151, 153  
most significant . . . . . 44, 78, 178, 191

## N

negative . . . . . 7-8, 10  
nybble . . . . . 11, 27

## O

octal . . . . . 128  
offset . . . . . 27-9, 48-9  
operand . . . . . 18, 26, 28-9, 33, 41, 139, 151-2  
operating system . . . . . 4, 15, 22, 147  
origin . . . . . 36, 179

## P

page . . . . . 21, 25  
parameter . . . . . 56  
PIA . . . . . 22, 156, 159, 162  
pixel . . . . . 159, 167  
pointer . . . . . 24, 28-9, 48  
POKE . . . . . 151  
port . . . . . 132, 156, 159  
position independent code . . . . . 27, 29, 168

postbyte	26-8, 132
post increment	29
precedence	54
pre decrement	29
program counter	56
pseudo-op	35-6, 153
push	52

<b>R</b>	
RAM	4, 16, 20-23, 127, 160
random	178, 180
registers	23-9, 31, 33-5, 45-6, 48, 53, 55, 128, 156, 199
REM	18
reset	40-1
return address	57
ROM	4, 16, 20-23
ROM routine	128, 161
rotate	145
rounding	12
routines	22

<b>S</b>	
SAM	4, 22, 159
scientific notation	8
screen memory	200
set	40-1, 45
shifts	44
speed	15-6
stack	24, 51-2, 55-6, 58
stack pointer	53-4, 56-7
status	24
status bit	42
subroutine	18, 55-6, 180, 199
sustain	192
syntax	148

<b>T</b>	
tapes	4
temporary storage	57
tone	191
top-down	148
truth table	40
two's complement	7-8, 28, 128, 136

<b>V</b>	
variables	4, 18
VDG	22, 156, 159-60

<b>W</b>	
warranty	20
waveform	191
workspace	57

<b>Z</b>	
zero page	27

6809	1,20
------	------





Write faster, more powerful space saving programs for your Dragon.

Written exclusively for Dragon users, Dragon Machine Language for the Absolute Beginner offers a complete instruction course in 6809 Machine Language, with particular reference to the Dragon 32.

Even with no previous experience of computer languages, the easy-to-understand 'no jargon' format of this book will enable you to discover the power of the Dragon's own language.

After introducing you to machine language and the 6809 instruction set, the book provides you with a series of short test programs which are designed to demonstrate all the Dragon's machine language instructions. These programs illustrate the use of the various instructions, their effects and actions, and will enable you to gain a practical understanding of machine language.

You are encouraged to develop routines with all the instructions in order to become familiar with the 6809's instructions as well as gaining experience in writing short machine language routines.

In addition to learning machine language, there are sample programs designed to demonstrate the power and potential of machine language in extracting the most from your Dragon. You will find that these programs contain routines that you can use when writing your own programs.

In logical steps the book takes you through a comprehensive course in machine language, including understanding assembly language, designing and writing your own programs and a thorough grounding in the purpose and use of each of the instructions in the 6809 Instruction Set.

**Melbourne House Publishers**  
ISBN 0 86161 130 6