HB211

PROGRAMMING THE 6800 MICROPROCESSOR

— Bob Southern —
Algonquin College
Ottawa Ont. Canada

A self-instructional workbook
for assembly language and machine code programming
of the 6800 family of microprocessors and peripherals

MOTOROLA Semiconductor Products Inc.

Acknowledgments

— Many people helped make this workbook possible. I would like to thank Peter Booler, Brian Bradley, Michel Brulé and Bill Foster of Algonquin College, and Don Lindsay of Dynalogic Limited for their advice and comments. I also would like to thank Lynne Hall who formatted and typed this book. Lastly, I would like to thank Richard Leir, John Oldfield and John Quarterman for their time in testing the final version of this book.

— The program on the front cover was written by Don Lindsay of Dynalogic Limited, Ottawa.

Bob Southern

Disclaimer

— The information contained in this workbook has been carefully checked and is believed to be correct. However the author and publisher cannot assume responsibility for errors or omissions or liability for any damages or consequential damages arising from the use of this workbook.

9 8 7 6
Printed in Canada

PROGRAMMING THE 6800 MICROPROCESSOR

ABOUT THIS WORKBOOK

This workbook has one purpose only, to help you to learn the fundamentals of assembly language and machine code programming of the 6800 microprocessor and its peripheral devices. Considerable coverage is given to programming of input/output devices, an essential part of microprocessor applications. The ACIA and PIA, each with their various modes of operations, are explored in detail in both non-interrupt and interrupt modes. Program design and documentation is emphasized, enabling others to understand the purpose and operational details of your programs. Programming hints and aids are included along with the answers.

FOR WHOM

This workbook was designed primarily for use by students at the community college level, although it has been successfully used by at least one capable high school student. Previous programming experience is not necessary. Early high school mathematics is adequate, although mathematical competence beyond this level is a good predictor of success.

THIS WORKBOOK IS AVAILABLE FROM

Motorola Semiconductor Products Inc.
Literature Distribution Center
P.O. Box 20924
Phoenix, AZ 85036

HOW TO USE THIS WORKBOOK

The programmed notes in this workbook are for your use at your own pace. Take your time, proceeding to the next frame when you are satisfied with your answer, after comparison with the answer given.

To use these notes effectively:

(a) Cover the given answer shown below the horizontal line following each question. A data card is very convenient for this.

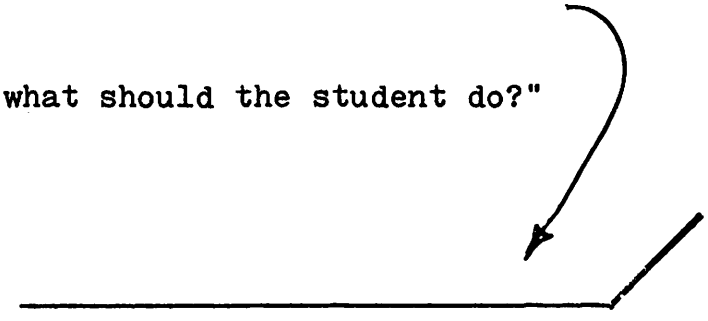
(b) Read the text material given in the frame.

(c) Write your answer to the question asked.

(d) Compare your answer with the answer given and when you fully understand any differences, if any, proceed to the next paragraph.

For practice attempt the following question, after covering the answer below the line. Write your answer here.

"After answering the question what should the student do?"



Answer: The student should compare his/her answer with the one given in the workbook and, when satisfied with any differences, move on to the next paragraph.

BINARY AND HEX NUMBERS

Before starting please read the left page to get the most benefit from this programmed instruction workbook.

PRE-TEST

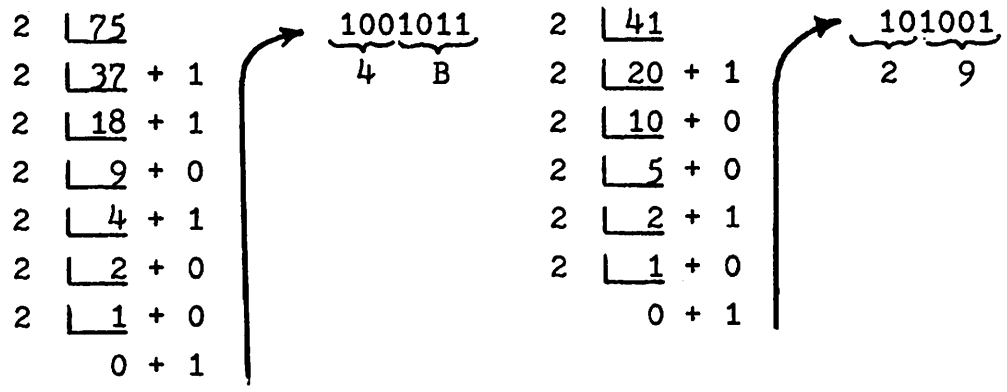
If you are familiar with binary and hexadecimal arithmetic operations, try the test below. If this is not familiar to you, turn the page and start the instruction in frame 1-1.

(a) Calculate 75 - 41, after first converting each decimal number to its hexadecimal value, then performing the subtraction. Verify by converting your answer back to decimal. Write your answer on this page.

(b) Repeat (a) in binary rather than hexadecimal. Solutions are on the next page.

Contd...

(a) Solution: 75 - 41 = 34 (decimal)



Calculate -29 then add 75, all in hex.

```
 FF
-29
 D6
+ 1
 D7
+4B
 22 hex
```

$2 \times 16^0 = 2$
$2 \times 16^1 = 32$
} = 34 decimal

(b) 75 = 01001011 (as an 8 bit number)

41 = 00101001

one's complement of 41 = 11010110

\+ 1

two's complement of 41 = 11010111

plus 75 01001011

1 00100010

overflow

$1 \times 2^1 = 2$

$1 \times 2^5 = 32$

34 decimal

If your answers are correct skip over to Chapter 2, otherwise start Chapter 1 instruction on the opposite page.

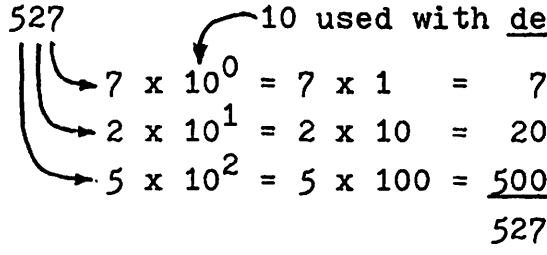
The number system most familiar to us is the decimal one, in which a character has ten possible states, 0 to 9. Adding 1 to 9 results in 10, that is "0" with "1 to carry" or simply "0 with a carry".

A decimal number 527 means:

	7 units	= 7
plus	2 tens	= 20
plus	5 hundreds	= 500
	Total	= 527

Another decimal concept to note is that $10^3 = 10 \times 10 \times 10 = 1000$. Similarly $10^2 = 10 \times 10$, $10^1 = 10$ and $10^0 = 1$. In fact any value, raised to the power of zero, equals 1.

The decimal number 527 may then be expressed as:



Computers use the binary or two-state number system, that is each "binary digit" or "bit" has only two states, 0 or 1. Adding 1 to 1 results in 0 with a carry.

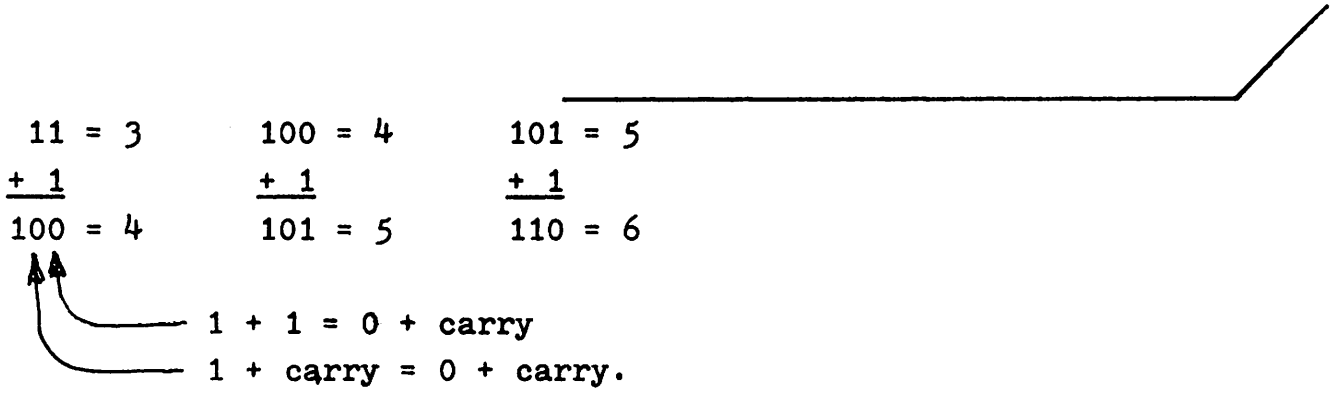
The first 3 numbers in the binary number system are 0, 1 and 10. This is seen by adding

0	then	1	
+1		+1	
=1		=10	= 2 (decimal)

In binary add 2 + 1. Your answer should be written above this line. Then check your answer.

10		
+ 1		
=11	= 3 (decimal)	11 (binary) = 3 (decimal)

Now calculate the binary values for 4, 5 and 6, starting from the binary equivalent of 3.



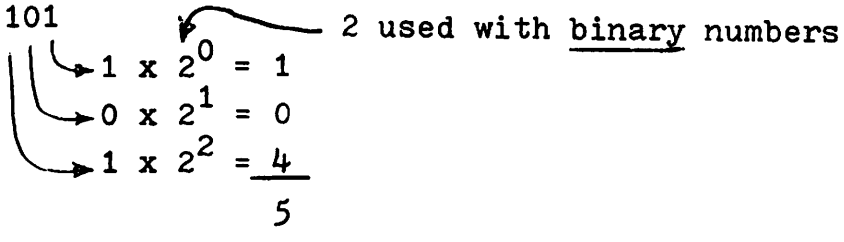
In summary the binary equivalents of 0 to 6 are:

Decimal	0	1	2	3	4	5	6
Binary	0	1	10	11	100	101	110

Leading zeros could be used with the above binary numbers, if desired, e.g., 110 = 0110 if a 4 bit number is required.

A subscript will be used from now on to denote the number system, e.g., 110_2 is the binary number 110, while 110_{10} is the decimal number 110. When the number system is obvious the subscript may be omitted.

Interpretation of the binary number 101 is:



1-3

Determine the binary value for 8 and 9.

8 = 1000
9 = 1001

To verify

			OR	
110	= 6			110 = 6
+ 1				+10 = 2
111	= 7			1000 = 8
+ 1				+ 1
1000	= 8			1001 = 9
+ 1				
1001	= 9			

The second solution is more direct and also demonstrates binary addition with a carry.

1-4

In the binary number 101, the right bit carries the least weight and is therefore called the Least Significant Bit or LSB. The left bit carries the most weight (2^2 in this case) and is the Most Significant Bit or MSB.

In binary, calculate 6 + 4. Verify by converting your answer to decimal.

6 = 110
+4 = 100
10 1010

$0 \times 2^0 = 0$
$1 \times 2^1 = 2$
$0 \times 2^2 = 0$
$1 \times 2^3 = 8$

10_{10}

Yes! It works.

1-5

Calculate 8 + 7 in binary. Verify your answer by converting it back to decimal.

$1111_2 = 15_{10}$

$$\begin{array}{r} 1000 = 8 \\ +0111 = 7 \\ \hline 1111 = 15 \end{array}$$

1111

$1 \times 2^0 = 1$

$1 \times 2^1 = 2$

$1 \times 2^2 = 4$

$1 \times 2^3 = 8$

15

In summary the binary equivalents for 0 to 15 are:

0000 = 0	0100 = 4	1000 = 8	1100 = 12
0001 = 1	0101 = 5	1001 = 9	1101 = 13
0010 = 2	0110 = 6	1010 = 10	1110 = 14
0011 = 3	0111 = 7	1011 = 11	1111 = 15

1-6

Each bit of a binary number is assigned a bit number which is the same as its binary exponent as shown below.

1011

bit #0

bit #1

bit #2

bit #3

What is another name for bit #3 in this binary number 1011?

MSB or Most Significant Bit.

The bit number is also useful in determining the weight of each bit in a binary number, e.g.,

bit #5

110110 — same.

1×2^5

Let's look at a method to convert from decimal to binary. This method involves successive division of the decimal number by 2, noting the remainder at each stage. Conversion of 19_{10} to binary is illustrated.

```
2 |19        <-- remainder
2 | 9 + 1
2 | 4 + 1
2 | 2 + 0
2 | 1 + 0
    0 + 1
```

10011. This is obtained by reading the remainders, bottom to top.

To verify: 10011

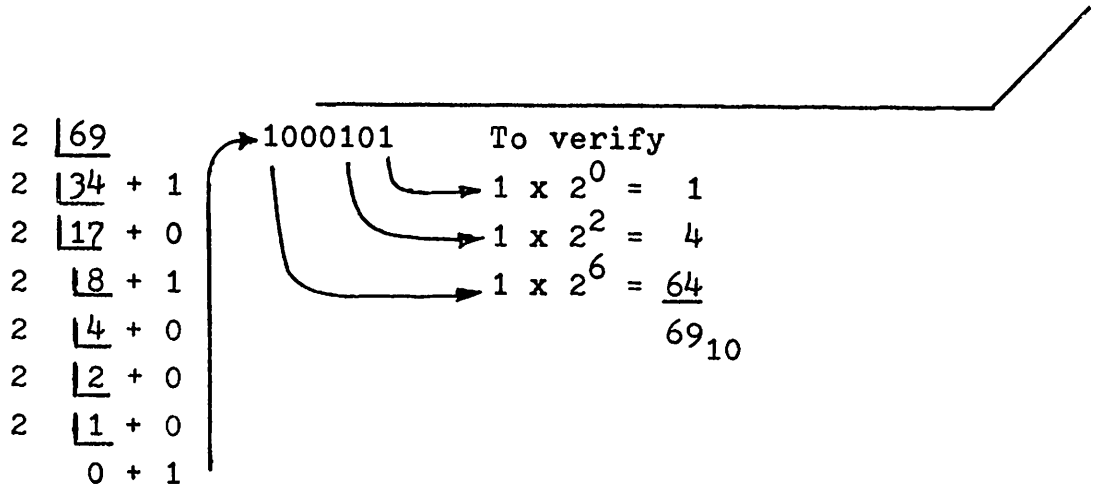
$1 \times 2^0 = 1$

$1 \times 2^1 = 2$

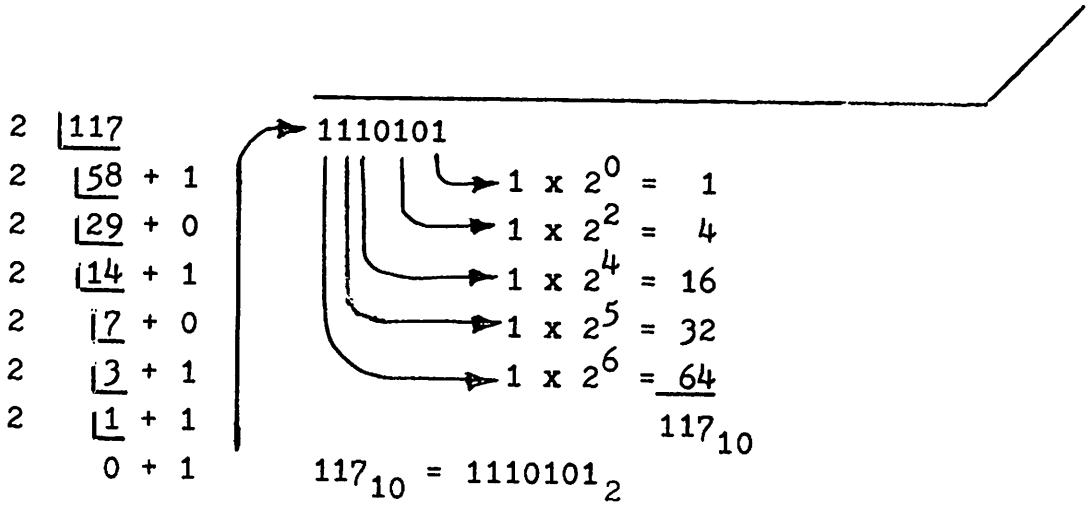
$1 \times 2^4 = 16$

19_{10}

Now calculate the binary equivalent of 69 and verify your answer.

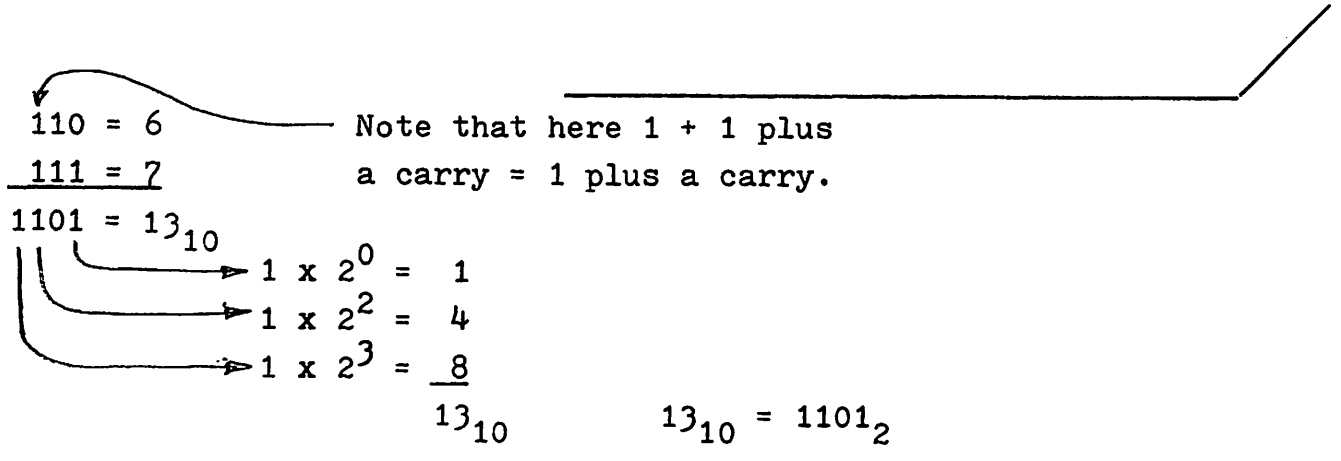


Convert 117_{10} to binary and verify your answer by reconverting to decimal.



If you are satisfied with your progress proceed to the next frame. If not, try another number of your own choice now.

Let's look at binary addition now. Add 6 + 7 in binary and verify your answer by converting it to decimal.



Calculate 5 + 7 in binary and convert your answer to decimal to verify it.

$5 = 101$

$7 = 111$

$12_{10} = 1100_2$

$1 \times 2^2 = 4$

$1 \times 2^3 = 8$

$\rightarrow 12_{10} = 1100_2$

Values less than 1 can be expressed in binary as in the example below

101.1 binary point

The 1 on the right side of the binary point carries the weighting of 2^{-1} (or 0.5_{10}), since the binary exponent continues to decrease by 1 for each move to the right. The decimal value is then

$1 \times 2^2 = 4$

$0 \times 2^1 = 0$

$1 \times 2^0 = 1$

$1 \times 2^{-1} = 0.5$

101.1 $\qquad 5.5$

Express 110.11 in decimal.

6.75_{10}

$1 \times 2^2 = 4$

$1 \times 2^1 = 2$

$0 \times 2^0 = 0$

$1 \times 2^{-1} = 0.5$

$1 \times 2^{-2} = 0.25$

110.11 $\qquad 6.75_{10}$

The weight of each bit of a binary number can be summarized by:

	1	1	1	1	1 .	1	1	1
Binary exponent or Bit # →	4	3	2	1	0	-1	-2	-3
Binary Value →	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
Decimal Equivalent →	16	8	4	2	1	1/2	1/4	1/8

We'll return to the binary number system later. Meanwhile let's look at another way to express binary numbers, in hexadecimal form (hex for short) meaning 16 possible states.

A 4 bit binary number has 16 possible states, 0000 to 1111. Expressing each of the first ten values as a single character is quite familiar now.

0000 = 0	0101 = 5
0001 = 1	0110 = 6
0010 = 2	0111 = 7
0011 = 3	1000 = 8
0100 = 4	1001 = 9

The problem now is that we need 6 more characters to express the next values, 1010 to 1111. Arbitrarily the letters A to F are assigned to express the missing values, that is:

A = 1010
B = 1011
C = 1100
D = 1101
E = 1110
F = 1111

The even values, A, C and E can be remembered by the word "ACE"

Appendix A summarizes the binary equivalents of the hex values, 0 to F. Without looking in Appendix A, what is the decimal equivalent of hex code E?

14_{10} $E_{16} = 1110 = 14_{10}$

By breaking up longer binary numbers into groups of 4 bits each we can express them in their hex equivalents e.g., the 8 bit binary number

10011010 can be grouped as

1001 1010
9 A or 9A as the hex equivalent.

Each of the 2 characters can then be a number (0 - 9) or a letter (A - F). Express 11000011 in hex and mark bit #6 of this binary number.

bit #6

C3 1100 0011 → 11000011
C 3 76543210 ← bit #

Hex codes are very popular with 8 bit microprocessors, such as the 6800, with 2 hex characters equalling 8 bits or 1 byte. If for some reason only 7 bits are used in a binary number, a leading zero may be added to fill out the 8 bits, e.g.,

1011101 = 01011101
5 D

Express each of the following binary numbers in hex:

11000101 1111000 111011

11000101 01111000 00111011
C 5 7 8 3 B

With a base of 16 the hex number 78 equals:

78 ← 16 used here for hex numbers.
8 x 16^0 = 8
7 x 16^1 = 112
120_{10}

The hex number 78 can be expressed as 78_{16} to avoid confusion with the decimal number 78_{10}, a different value.

1-15

Express each of the following hex numbers in binary and in decimal:

D4

39

6A

D4 = 11010100
D 4

D4
4 x 16^0 = 4
D = 13 13 x 16^1 = 208
212_{10}

39 = 00111001
3 9

39
9 x 16^0 = 9
3 x 16^1 = 48
57_{10}

6A = 01101010
6 A

6A
A = 10 10 x 16^0 = 10
6 x 16^1 = 96
106_{10}

1-16

Addition in hex can be challenging, although the problem does not exist for computers since they work in binary. Hex is for our convenience in expressing binary numbers.

One solution is to convert to binary, add the numbers and convert the answer back to hex, possible but not the fastest way. If we had 8 toes on each foot we could count on our toes to add. Did you ever consider why our number system has a base of ten?

The solution proposed is the use of the number line, until you become more familiar with hex addition.

For example: 9 + 3 = C

start here — 1 2 3 — count 3 to the right to get "C"

0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F

Going beyond F produces a carry

e.g., D + 5 = 12_{16}, that is 2 plus a carry.

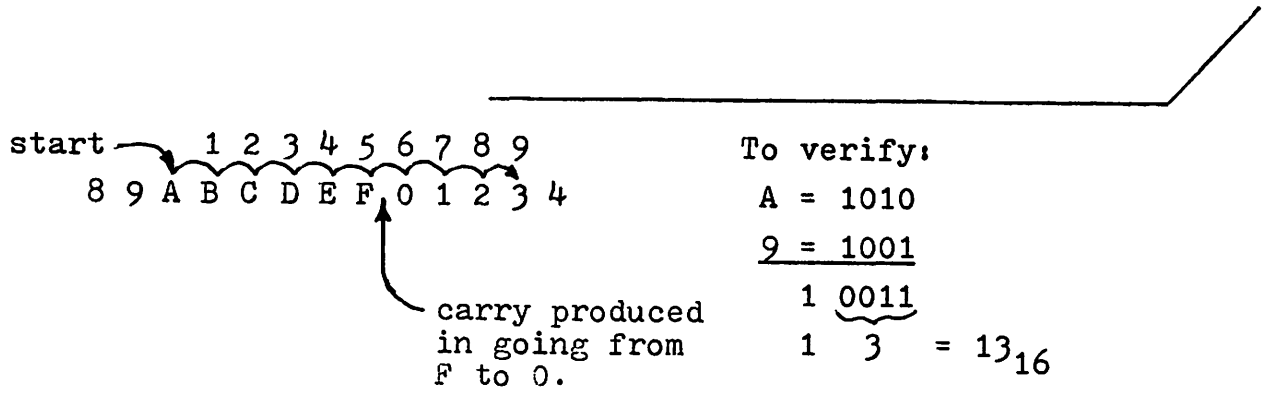
start — 1 2 3 4 5

0 1 2 3 4 5 6 7 8 9 A B C D E F 0 1 2 3 4 5 6 7 8 9 A B C D E F

F to 0.

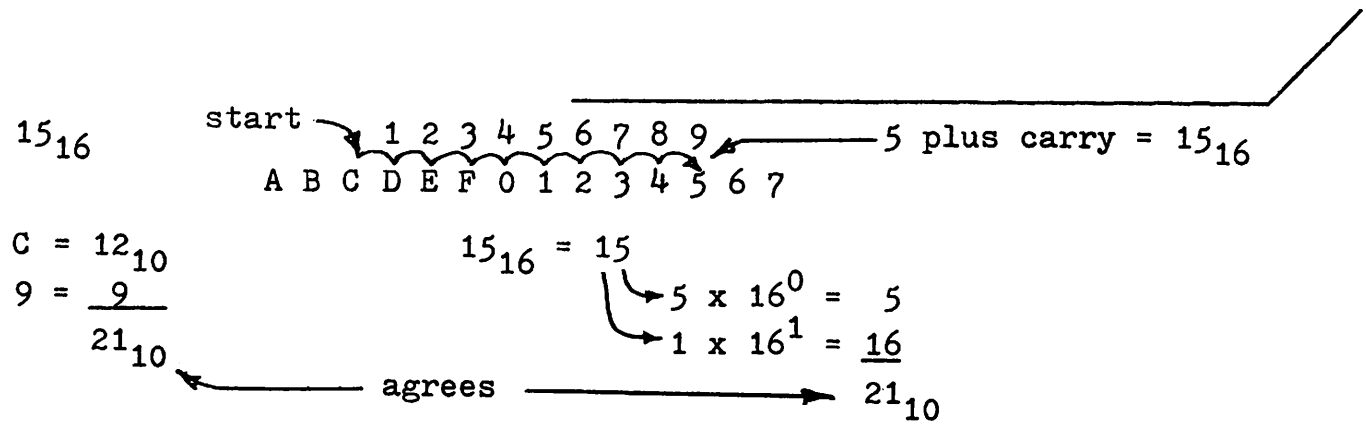
Contd...

Using this principle show that A + 9 = 13_{16}.



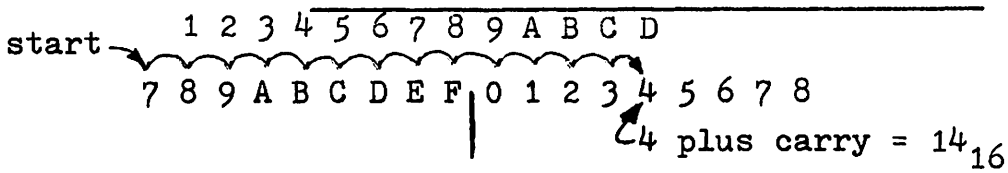
1-17

Now add C + 9 and verify your answer by adding the decimal equivalents.



Now add 7 + D and verify your answer by adding in decimal.

14_{16}



To verify 14_{16}

$4 \times 16^0 = 4$ $\qquad$ $7 = 7$

$1 \times 16^1 = \underline{16}$ $\qquad$ $D = \underline{13_{10}}$

20_{10} $\qquad$ 20_{10}

20_{10} ← agrees → 20_{10}

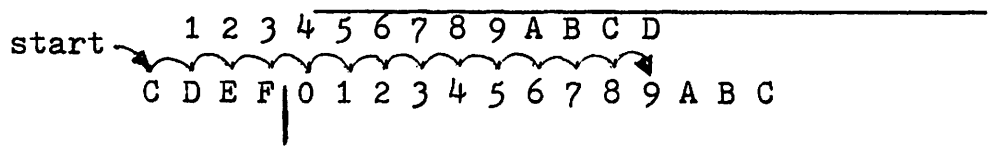
It would have been easier to add 7 to D rather than D to 7. The answer still is 14_{16}.

Add the hex numbers C and D. Verify your answer.

C

+D

19_{16}



To verify 19_{16}

$9 \times 16^0 = 9_{10}$ $\qquad$ $C = 12_{10}$

$1 \times 16^1 = \underline{16_{10}}$ $\qquad$ $D = \underline{13_{10}}$

25_{10} ← agrees → 25_{10}

To add 2 column hex numbers each column is added separately, as in decimal. If the right column produces a carry it is added to the left column

e.g.,
```
  2F
+13
  42
```
F + 3 = 2 plus carry

2 + 1 + carry = 4

Add the hex numbers 3E + 27.

```
  3E
+27
  65
```
E + 7 = 5 plus carry

3 + 2 + carry = 6

Add the hex numbers 4D and 25.

72_{16}

```
  4D
  25
  72
```
D + 5 = 2 plus carry

4 + 2 + carry = 7

To verify we'll convert all data to decimal

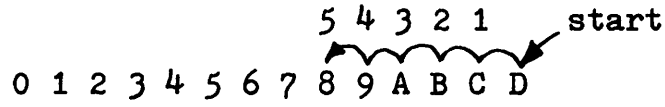
$4D_{16} = 4 \times 16^1 + 13 \times 16^0 = 64 + 13 = 77_{10}$

$25_{16} = 2 \times 16^1 + 5 \times 16^0 = 32 + 5 = 37_{10}$

$72_{16} = 7 \times 16^1 + 2 \times 16^0 = 112 + 2 = 114_{10}$

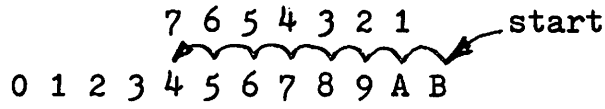
```
  77
+ 37
 114
```
114_{10} ← agrees →

Subtraction involves moving to the left on the number line, e.g., D - 5 = 8 as seen below

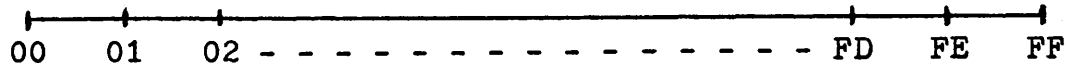


For the moment we will avoid "borrow" operations.
Calculate B - 7.

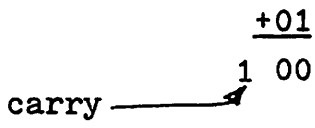
4



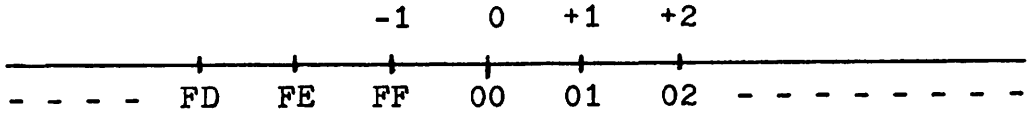
If we are to handle subtraction we have to recognize negative numbers since 9 - 3 is actually 9 + (-3). Consider the number line for an 8 bit binary number. Expressed in hex it extends from 00 to FF (0 to 255_{10})



However, if 1 is added to FF the result, still using 2 hex characters (8 bits), is



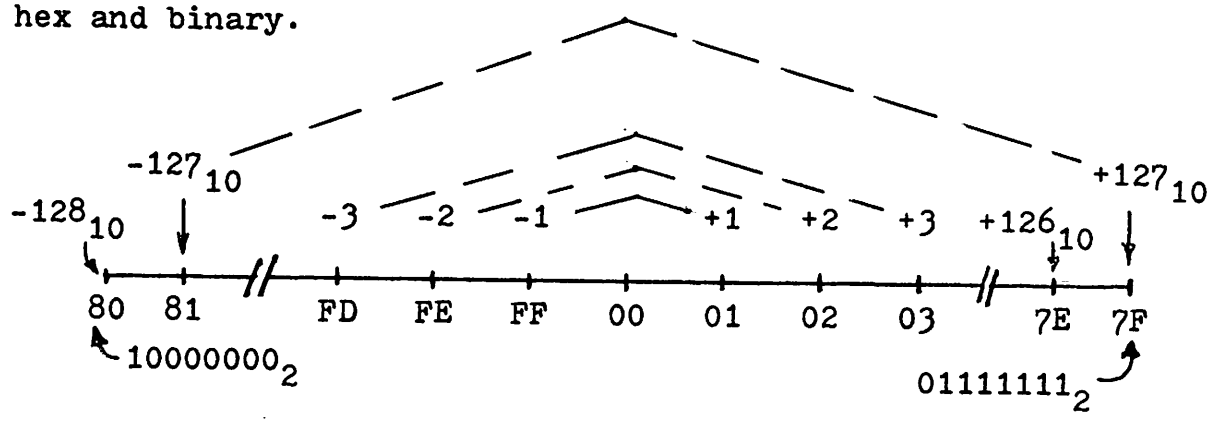
or 00, the carry being lost as an overflow, outside the 8 bit limit. The question now asked is "What number, when 1 is added to it, becomes 0?" The answer is -1. By definition therefore FF = -1. We now reconstruct our number line



What is the value of FD based on this number line?

-3 Since FD + 3 = 00 (carry is outside the 8 bit limit)
This new number line is called a signed number line since it permits both positive and negative values.

Continuing with the signed number line if the leading bit (MSB) of the 8 bit number = 1, that is 8 or more for the first hex character, the number by definition is negative. The extent of this signed number line is shown below in decimal, hex and binary.



The extent of this signed number line is then -128_{10} to $+127_{10}$. Based on this number line which of the following hex values are negative,

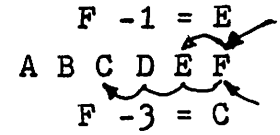
7A 94 F2 00 8E CA

All except 7A and 00 are negative, having a leading hex character 8 or larger. If converted to binary all except 7A and 00 would have 1 as a leading bit.

If a larger range is needed for the signed number line 16 bits (2 bytes) could be used, again providing negative values if the leading bit equals 1. This is sometimes referred to as a double precision value.

To determine the negative value for the hex number 31 is more difficult. A procedure shown below is based on the 2's complement arithmetic used in binary subtraction.
The procedure then is:

- Start with the largest possible hex value (ignoring the sign) ⟶ FF
- then subtract the number ⟶ -31
  using the number line approach CE
- then add 1 ⟶ + 1
  CF now equals -31_{16} CF



To prove it the sum of CF and 31 should be zero in 2 character hex format. Prove it.

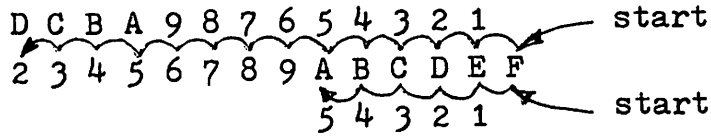
```
 CF
+31
100
```

F + 1 = 0 + carry
C + 3 + carry = 0 + carry
carry, which is ignored as an overflow

CF = -31_{16}

Determine the hex value for -5D and prove that it is correct by adding +5D to it.

```
 FF
-5D
 A2
+ 1
 A3 = -5D
```



In the top row a more direct subtraction is seen in that F and D are separated by 2, hence F -D = 2.

```
To check   A3
          +5D
carry ⟶  1 00
```

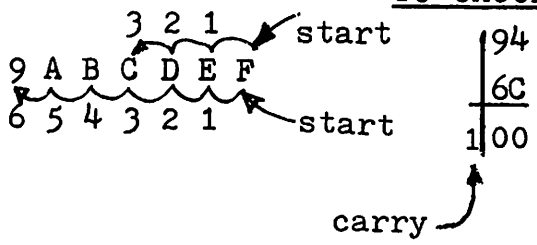
Now calculate -6C and verify it.

-6C = 94

```
  FF
 -6C
  93
+ 1
  94
```

To check:

```
  94
 +6C
100
```



The "two hex character" value of -3 is FD. If 4 characters are used to express -3, prove that -3 = FFFD.

```
 FFFD
+  3
10000
↑ carry.
```

Similarly a 6 character representation would be FFFFFD.

To determine the value of -3 using 4 hex characters, the procedure is

```
 FFFF
-  3
 FFFC
+  1
= FFFD
```

Using 6 hex characters -3 equals

```
 FFFFFF
-     3
 FFFFFC
+     1
 FFFFFD
```

Almost all our work will employ 2 hex characters only. For 6 hex characters (3 bytes) the signed number line would extend from 800000_{16} (most negative) to $7FFFFF_{16}$ (most positive).

We now have the capability to subtract in hex since 72 -3D is actually 72 +(-3D). Once -3D has been calculated the hex addition will produce the answer. Try it.

```
 FF   largest hex value
-3D
 C2
+ 1   plus 1
 C3   = -3D
+72   now add the 72
1 35  answer
  └overflow ignored
```

To check:
If 72 -3D = 35 then 35 + 3D = 72

```
 35
+3D
 72
```

To verify further we will convert all data to decimal:

$72 = 7 \times 16^1 + 2 \times 16^0 = 112 + 2 = 114_{10}$

$3D = 3 \times 16^1 + 13 \times 16^0 = 48 + 13 = 61_{10}$

$114_{10} - 61_{10} = 53_{10}$

$35 = 3 \times 16^1 + 5 \times 16^0 = 48 + 5 = 53_{10}$ ← agrees

Let's try one more subtraction. Calculate E3 -DC.

```
 FF
-DC
 23
+ 1
 24 = -DC
+E3
 07
```

E3 is already a negative number

$E3 = -1D_{16} = -29_{10}$

```
 FF
-E3
 1C
+ 1
 1D
```

DC is already a negative number too

$DC = -24_{16}$

Therefore $-DC = 24_{16} = 36_{10}$

```
 FF
-DC
 23
+ 1
 24
```

To verify: E3 - DC = 07

OR -29 - (-36) = 7

This shows that subtraction is valid with positive negative or mixed numbers. Errors will occur if the result goes beyond the range of -128_{10} to 127_{10}, the limit of an 8 bit signed number.

Now calculate 57 -2C and verify your answer in decimal.

```
 FF
-2C
 D3
+ 1
 D4
+57
1 2B
```

To check $57_{16} = 5 \times 16^1 + 7 \times 16^0 = 80 + 7 = 87_{10}$

$2C = 2 \times 16^1 + 12 \times 16^0 = 32 + 12 = 44_{10}$

Total 43_{10}

$2B = 2 \times 16^1 + 11 \times 16^0 = 32 + 11 = 43_{10}$

As a variation, let's reverse the data in the last question. Calculate 2C -57.

D5 or -2B

```
 FF
-57
 A8
+ 1
 A9
+2C
 D5
```

But D5 is a negative number. To find its positive equivalent:

```
 FF
-D5
 2A
+ 1
-2B
```

Therefore D5 = -2B, the same answer but the opposite sign, compared to the previous question, since the data was reversed.

To complete this section let's review it all within several questions. Given two decimal numbers, 47 and 73, calculate the sum by converting to hex, adding, then converting back to decimal. Verify by decimal addition.

$$\begin{array}{r} 47 \\ +73 \\ \hline 120_{10} \end{array}$$

2 | 47	
2 | 23	+1
2 | 11	+1
2 | 5	+1
2 | 2	+1
2 | 1	+0
0	+1

$101111 = \underbrace{0010}_{2}\underbrace{1111}_{F}$

2 | 73	
2 | 36	+1
2 | 18	+0
2 | 9	+0
2 | 4	+1
2 | 2	+0
2 | 1	+0
0	+1

$1001001 = \underbrace{0100}_{4}\underbrace{1001}_{9}$

$$\begin{array}{r} 2F \\ +49 \\ \hline 78 \end{array}$$

$8 \times 16^0 = 8$

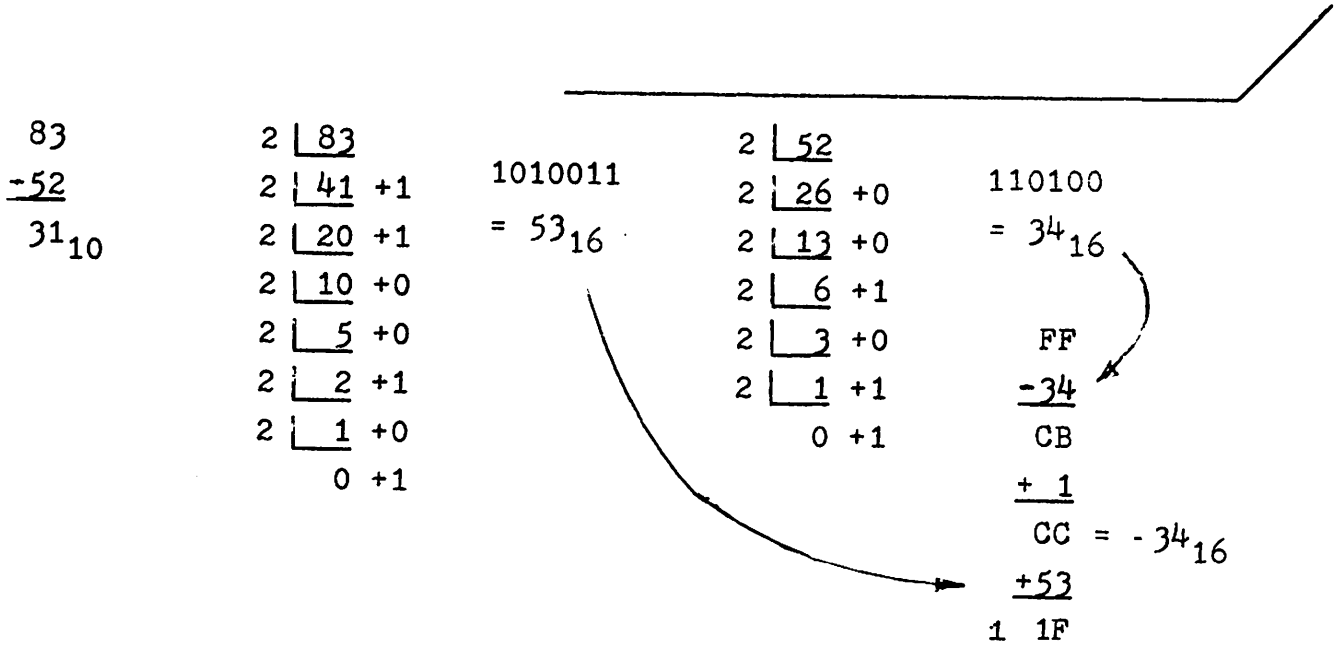
$7 \times 16^1 = 112$

120_{10}

start → 1 2 3 4 5 6 7 8 9

F 0 1 2 3 4 5 6 7 8

Now perform the following decimal subtraction 83 -52 by converting to hex, subtracting, then converting to decimal. Verify in decimal.



$1F = 1 \times 16^1 + 15 \times 16^0 = 31_{10}$ At last! It agrees.

Binary subtraction is not essential if you can subtract in hex. However it is included to complete the arithmetic operations in both formats. From a previous hex example,
D -5 = 8

```
D = 13 = 1101      1101
5 =  5 = 0101     -0101
                   1000
```

As in hex subtraction start with the number to be subtracted, 0101 in this example. Complement it , that is each 0 becomes 1 and each 1 becomes 0. Then add 1. This will produce the negative value of the original number (-5 = 1011 below).

```
          0101
becomes   1010
plus 1    + 1
        = 1011   = -5
Now add the      +1101    +13
minuend 1101   1 1000  =    8
```

overflow or carry is ignored.

This subtraction is limited to 4 bits as shown above.
Now calculate 12_{10} - 7_{10} in binary.

```
12(10) = 1100      -7 = 1000        1100 = 12
 7(10) = 0111         + 1          +1001 = -7
                     1001         1 0101 =  5
```


Perform the following 8 bit subtraction:

```
 11010111     (215 decimal)
-10110100     (180 decimal)
```

Contd...

```
                10110100    = 180(10)
complemented =  01001011
plus 1          +       1
                01001100    = -180(10)
               +11010111      +215(10)
               1 00100011      35(10)
                 overflow
```

If your data is in hex form already it is more direct to subtract in hex. If the data is in decimal and conversion has to be made to binary first, it is your choice whether you subtract in binary or hex. If the answer is needed in hex, then hex is preferred.

1-37

Here is the last question for this chapter. Calculate in binary.

```
  10110100
- 11010111
```

11011101 which equals -35_{10}.

This is the previous question with the order reversed.

e.g., $180_{10} - 215_{10} = -35_{10}$

```
Details are:    11010111    (215(10))
                00101000    (complemented)
                +       1
                00101001    (two's complement) = -215(10)
                +10110100   (+180(10))
                11011101    (which is a negative answer)
```

To calculate its positive value:

```
                11011101
                00100010
                +       1
                00100011 = 35(10)
```

Therefore the answer $11011101_2 = -35_{10}$

ACCUMULATOR OPERATIONS

The 6800 microcomputer is capable of a simple task such as the addition of two numbers or a complex task such as the control of a piece of electronic equipment. In both cases the task is defined by a series of instructions to the computer, usually referred to as a program.

Many program formats exist, the most fundamental being machine code in which a series of 8 bit words are entered in the computer via switches on the front panel of the computer.

The next level up is the expression of each instruction as 2, 4 or 6 hex characters, permitting entry via a keypad which has one key for each hex character. This still is a form of machine code.

For longer programs it is very tedious to generate hex codes for each machine language instruction. The solution is to write the program in assembly language, in which each instruction is in an abbreviated English format. The computer itself then converts this assembly language program to machine code, using a ready-made program called an assembler.

Higher still in the hierarchy of program formats are languages like BASIC, oriented to mathematical calculations in which algebraic-like statements, including trigonometric functions, are interpreted into many bytes of machine code for execution by the computer.

Our interest in this workbook is in assembly language and machine code programs which link the computer to keyboards, printers, displays, communication devices and external electronic instruments.

Within the 6800 microprocessor (computer without memory or interfaces to external equipment) there are two "accumulators", A and B. Within each accumulator 8 bits of data can be added, subtracted or modified via many different arithmetical and logical operations.

One of the simplest assembly language instruction is "CLR A", formed from "CLeaR accumulator A', meaning "put a zero in each of the 8 bits of accumulator A." The machine code for CLR A, expressed in hex, is 4F. (You don't have to remember the machine code.)

Write what you think is the assembly language instruction to clear accumulator B.

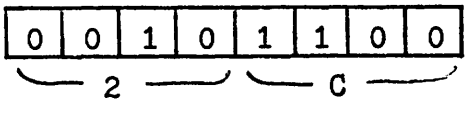
CLR B, which in machine code is 5F. This instruction can be written CLRB, omitting the space. Similarly CLR A can be written CLRA. Machine codes for all assembly language instructions are provided in Appendix C, at the end of this workbook. Instructions involving accumulators are on the first page of Appendix C.

2-2

If a hex value such as 2C is to be loaded into accumulator A the instruction is

LDA A #$2C (LDA A = LoaD Accumulator A).

The # symbol denotes that data follows immediately within the instruction. The $ symbol denotes that the data is in hex format. After this instruction is executed, the contents of ACC A is



since the LDA A instruction overwrites any previous contents of ACC A.

The instruction LDA A #$2C is formed of 2 parts:

LDA A (called the operator)which tells what happens (loading of ACC A),

#$2C (called the operand) which provides the data to be loaded.

Contd...

Such an instruction requires 2 bytes of machine code. LDA A, when followed by the # symbol is known as an immediate mode instruction; its machine code, 86, is found under the "IMMED" column, opposite LDAA in Appendix C. The second byte of the instruction contains the data to be loaded, 2C. Hence 86 2C = LDA A #$2C. Write the assembly language instruction and machine code to load ACC B with the hex value 7D.

LDA B #$7D C6 7D

Appendix G summarizes the use of special symbols such as # and $.

2-3

Write the instruction to load ACC A with the hex value 4D. Also write the machine code.

LDA A #$4D 86 4D

↑86, the machine code for the "operator" part of the instruction is also known as an operation code, commonly called the "op code".

The operand value, 4D, is also the code for the letter M, based on the ASCII (American Standard Code for Information Interchange) code, listed in Appendix B at the back of this workbook.

For practice use this table now to confirm that the ASCII code for Z is 5A, under column 5 opposite row A.

A spare copy of the Instruction Set is provided at the end of this workbook. It may be convenient to cut out this sheet, for use with each problem, instead of continually looking in the appendices.

2-4

Write the assembly language instruction and machine code to load ACC A with the ASCII code for the number 8. See Appendix B.

LDA A #$38 86 38

↖ from Appendix B - ASCII codes.

The ASCII codes for the numbers 0 to 9 are easy to remember, being 30 + N where N = 0 to 9.

2-5

Another form of the immediate instruction to load an ASCII code is seen in

LDA A #'Z (note the apostrophe)

in which the apostrophe denotes that the ASCII code for the letter Z is to be loaded. Hence the computer on assembling (converting to machine code) the above instruction automatically provides the desired ASCII code for the second byte of the machine code instruction. The resultant machine code is still 86 5A since this is still an immediate mode instruction. Such an instruction in which the computer provides the appropriate code for the desired character is often referred to as a "literal" instruction.

Write the literal instruction and the resultant machine code to load ACC B with the ASCII code for the number 7.

LDA B #'7 C6 37

↖ opposite LDAB under IMMED in Appendix C

2-6

Now write two instructions, the first to load ACC A with the hex value OF, the second to load ACC B with the ASCII code for the letter F (using a literal). For each instruction provide the machine code on the left side of the assembly language instructions.

```
86 0F          LDA A  #$0F
C6 46          LDA B  #'F
```

The first instruction loads a hex value, OF, into ACC A. The second loads an ASCII code for the letter F into ACC B. If the difference is not clear, please reread the question and answer.

If the above two instructions were executed in the order listed ACC A would take on a value, OF, and ACC B a value of 46. This example although trivial shows the beginning of a program, a series of instructions executed by the computer which modifies the contents of an accumulator or a memory location (discussed later).

2-7

Write the assembly language instructions to load ACC A with the ASCII code for A and load ACC B with the hex value OA. For each provide the machine code.

```
86 41          LDA A  #'A      OR LDA A #$41
C6 0A          LDA B  #$0A
```

Again note the distinction between a hex value and an ASCII code.

The above machine code and instructions are part of an assembler listing, the printout produced by the assembler when converting assembly language instructions to machine code.

The addition of 2 hex values, 3F and 27, in ACC A can be performed by

```
4F       CLR A
8B 3F    ADD A   #$3F   (Adds 3F + 0 = 3F in ACC A)
8B 27    ADD A   #$27   (3F + 27 = 66 in ACC A)
```

$_{16}$

machine code | assembly language instructions

Rewrite the above, using 2 rather than 3 instructions, again providing the machine code.

```
86 3F          LDA A   #$3F
8B 27          ADD A   #$27
```

This method is preferable to the one above since it is shorter.

The memory of a computer, where data is stored, can be envisaged as a series of mail boxes, each with a 4 character hex address, e.g. 14D5, and the capability to store one byte of data. The instruction

LDA A $12B7 (no # this time)

loads ACC A with the 8 bit contents of address 12B7, without destroying the contents of 12B7. Such an instruction is known as an EXTENDED mode instruction, requiring one byte for the operator (LDA A) and 2 bytes for the operand ($12B7). Hence LDA A $12B7 becomes B6 12B7. The B6 is found under the EXTND heading, opposite the LDAA instruction in Appendix C. The total number of bytes required (3) is found two columns to the right of B6, under the # column.

Contd...

2-9
Contd.

Write the assembly language instructions and machine code to load accumulator B with the contents of address 06E4.

```
F6 06E4          LDA B  $06E4
```

If address 06E4 contains 3F then ACC B will contain 3F after execution of this instruction. In the above instruction LDA B is the operator while 06E4 is the operand, denoting the data source.

2-10

Write the assembly language instructions to add the contents of memory addresses 1C00, 1C01 and 1C02, the answer residing in ACC B. Provide the machine code.

```
F6 1C00          LDA B  $1C00
FB 1C01          ADD B  $1C01
FB 1C02          ADD B  $1C02
```

2-11

The accumulators are used for many purposes within a program. Data, after being processed in an accumulator, usually is stored in a memory location, e.g.,

STA A $064C

which stores the contents of ACC A in address 064C but does not destroy the contents of ACC A. This instruction, referencing a 4 character hex address, also is "extended" mode. Write the machine code for the above instruction.

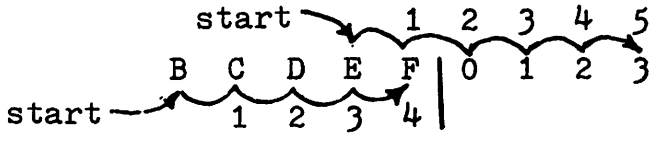
```
B7 064C
   |
   address
   STA A (extended mode)
```

2-12

Write the assembly language instructions and machine code to add the hex contents of addresses 14D0 and 14D1, then store the sum in address 14D2, without using ACC A.

If 14D0 contains 3E (14D0/3E) and 14D1 contains B5 (14D1/B5), what will the hex value in address 14D2 be when this program is executed?

```
F6 14D0        LDA B  $14D0  (ACC B/3E)
FB 14D1        ADD B  $14D1  3E + B5 = F3
F7 14D2        STA B  $14D2  14D2/F3   (ACC B still contains F3)
```




```
  3E
+B5
  F3
```

2-13

To place a particular value in a particular memory address it is first necessary to set it into ACC A or B. With this in mind write the assembly language instructions and machine code to put the hex value 3B in address 12E3.

```
86 3B          LDA A  #$3B  }
B7 12E3        STA A  $12E3 }— assuming use of ACC A.
```

Such a procedure is known as initialization, providing a particular memory address with an initial value, for use during a program.

2-14

Write the assembly language instructions and machine code to initialize address 0439 with the ASCII code for the letter G, with the computer providing the ASCII code.

```
86 47          LDA A  #'G
B7 0439        STA A  $0439
```

Again it is not necessary to memorize the machine code for the instructions. However, the 86 and B7 values will soon become quite familiar.

2-15

The instruction SUB A $1524 subtracts <u>from</u> accumulator A the contents of address 1524. Write the assembly language instructions and machine code to:

(a) ADD the contents of addresses 13C4 and 13C8
(b) then SUBTRACT from this the contents of address 13CA
(c) then STORE the result in address 13CC.

```
B6 13C4   LDA A  $13C4 )
BB 13C8   ADD A  $13C8 ) —assuming use of ACC A
B0 13CA   SUB A  $13CA )
B7 13CC   STA A  $13CC )
```

An instruction which will produce the negative value of the contents of ACC A is

NEG A (NEGate accumulator A).

If ACC A contained 04 before execution of NEG A it would contain FC (-04) after execution. The machine code or operation code (op code) is 40 as seen in Appendix C opposite the 2's complement (Negate) instruction.

Like the CLR A instruction NEG is under the INHERent column, being complete within itself; that is it does not require another byte for the operand.

Write the assembly language instructions and machine code to store the value -3C in address 095A.

```
86 3C          LDA A  #$3C
40             NEG A
B7 095A        STA A  $095A
```

Address 095A now contains C4 (-3C)

Memory addresses referenced in an instruction normally require 2 bytes (4 hex characters) to describe them, e.g., LDA A $12A6, requiring an EXTENDed mode instruction. Memory addresses below 100_{16} require only 1 byte to describe them, as is seen in a DIRECT mode instruction, e.g.,

LDA A $4A

which loads ACC A from address 004A. The machine codes for DIRECT mode instructions are in Appendix C. For the above instruction the machine code is

96 4A

96 — LDA A (DIRECT mode)

4A — address 004A

Contd...

Write the instruction to store ACC B in address 66 using a DIRECT mode instruction. Write its machine code.

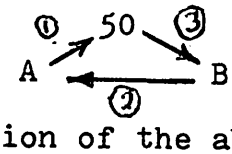
D7 66 STA B $66

Aside from requiring fewer memory locations to store the instruction a DIRECT mode instruction requires fewer machine cycles to execute as seen in Appendix C. Large programs often use addresses below 100 as a "scratch pad" storage area, e.g., for storage of counter values, or temporary storage of a byte of data. Use of this area of memory saves memory bytes and reduces execution time.

2-18

The instruction TAB transfers the contents of ACC A to ACC B. Similarly TBA provides the reverse transfer. Using as few instructions as possible, swap the contents of the two accumulators. Memory addresses below 100_{16} are available (use DIRECT mode only). Write the assembly language instructions and machine code.

```
97 50   STA A  $50 (or your choice of address)
17      TBA
D6 50   LDA B  $50 (or your choice of address)
```



Counter-clockwise execution of the above flow diagram would utilize TAB (op code 16).

Accumulator A can be incremented (1 is added to it) via the instruction

```
INC A    (INCrement accumulator A)
```

for which the op code is 4C.

Similarly DEC A (DECrement accumulator A) will decrease its contents by 1. Its op code is 4A. Accumulator B also can be incremented or decremented.

Calculate the contents of each accumulator after the following instructions are executed:

```
CLR A
CLR B
INC B
ADD A   #$2C
ADD A   #$16
TAB
NEG A
INC A
```

		ACC A	ACC B
CLR A		0	-
CLR B		0	0
INC B		0	1
ADD A	#$2C	2C	1
ADD A	#$16	42	1
TAB		42	42
NEG A		BE	42
INC A		BF	42

ACC A/BF

ACC B/42

```
  FF
 -42
  BD
 + 1
  BE
```

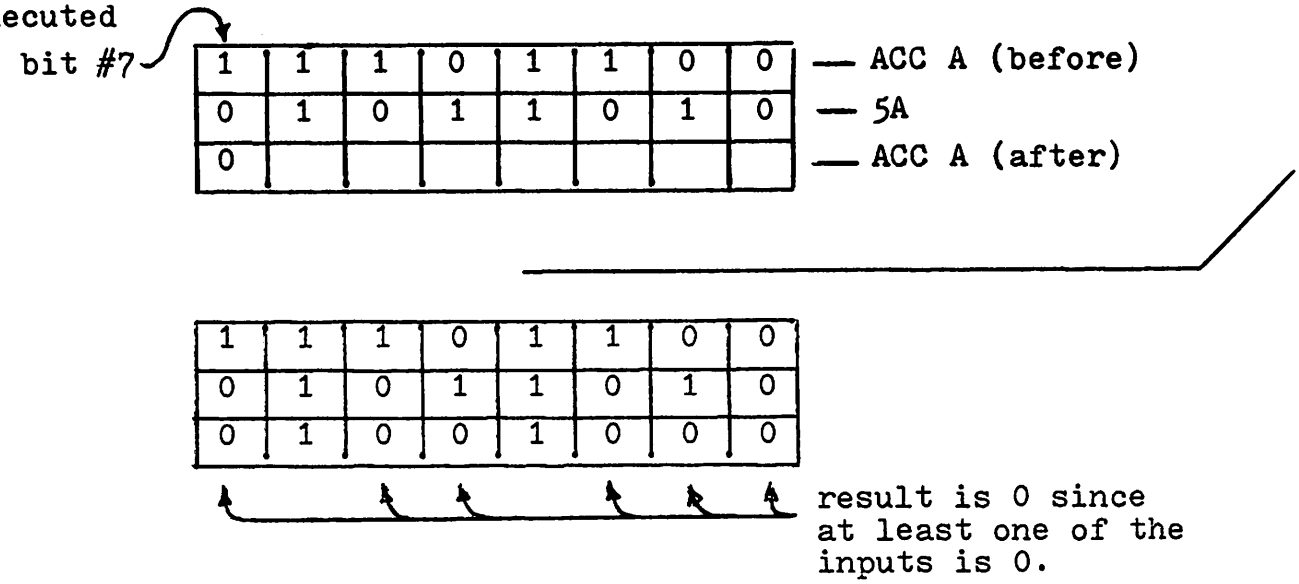
Therefore -42 = BE

Sometimes it is necessary to clear (force to 0) or set (force to 1) specific bits of an accumulator, without disturbing the other bits of the accumulator. This is accomplished via the AND and ORA operating on the accumulator. The AND instruction clears specific bits while the ORA instruction sets specific bits. The instruction

AND A #$5A (machine code 84 5A)

performs the "logical AND" operation (not addition) bit by bit with ACC A and the data 5A being inputs and ACC A holding the result.

In the "logical AND" operation each bit of the result will be 1, if and only if <u>both</u> the corresponding inputs are 1. Looking first at bit #7, below, one of the two inputs has a zero. Therefore bit #7 of the result is zero. Complete the bottom line showing the contents of ACC A after the AND A #$5A instruction is executed



If address 14A2 contains 7C, what will ACC A contain after execution of

```
LDA A  $14A2
AND A  #$BF
```

3C

		7	6	5	4	3	2	1	0	← bit #
7C	=	0	1	1	1	1	1	0	0	Contents of 14A2 to ACC A
BF	=	1	0	1	1	1	1	1	1	
		0	0	1	1	1	1	0	0	= 3C

Bit #6 is guaranteed to be zero regardless of the contents of address 14A2 since the "mask word", BF contains a zero in bit #6. The result can be shown as

X 0 X X X X X X

where X denotes the original data in ACC A before the AND operation. If the purpose of this operation was to clear bit #6 of the data in address 14A2, the modified data would then be stored back in address 14A2 by another instruction, STA A $14A2.

Write the assembly language instructions and machine code to clear bit #3 of the contents of address 1256.

Contd...

```
B6 1256      LDA A  $1256
84 F7        AND A  #$F7
B7 1256      STA A  $1256
```

7	6	5	4	3	2	1	0 ← bit #	
X	X	X	X	X	X	X	X	Contents of 1256
1	1	1	1	0	1	1	1	
X	X	X	X	0	X	X	X	{ ACC A (after) and contents of 1256 (after).

X represents undisturbed data

Bit #3 = 0 since X . 0 = 0 (symbol for logical AND) X could be 0 or 1

If X = 0, then 0.0 = 0

If X = 1, then 1.0 = 0

Therefore X.0 = 0

All other bits are unchanged since

X.1 = X If X = 1, then 1.1 = 1

If X = 0, then 0.1 = 0

Therefore X.1 = X (same as before)

2-23

Similarly all bits, except a specific bit, of a particular address can be cleared by the appropriate "mask word". Write the assembly language instructions and machine code to clear all bits, except bit #6, of address 065E.

```
B6 065E      LDA A  $065E
84 40        AND A  #$40
B7 065E      STA A  $065E
```

← bit #6 (40 = 01000000)

Since only bit #6 of the mask word = 1, then only bit #6 of the original contents of 065E will be retained. All other bits of the result will be zero. This technique will be used extensively later in this workbook.

The above AND instruction could be rewritten in terms of the binary value of the mask word e.g.,

```
AND A #%01000000
```

The % symbol indicates that a binary value will follow. This form is often useful to both the programmer and the user in quickly determining which bits are cleared.

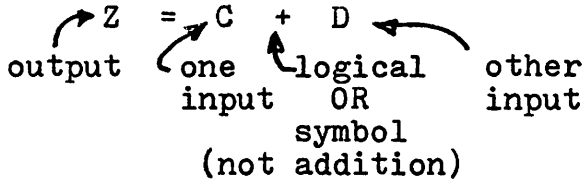
An ASCII code, produced by an external device, such as a keyboard requires only 7 bits to describe it. The 8th bit (bit #7) may be 1 or 0 depending on the particular data source. Assume that an ASCII code is now in ACC A. Write the assembly language and machine code instruction to clear bit #7 of the ASCII data. Use the binary version of the mask word in your answer.

```
84 7F          AND A  #%01111111
```

Note that the machine code instruction is still expressed in hex even though the assembly language instruction uses a binary mask word.

In summary a 0 is used in the mask word of an AND operation for each bit that is to be cleared. All other bits of the mask word are 1.

We have seen how to clear specific bits. Let's look at a method to set specific bits. For this purpose the "logical OR" operation is used (sometimes called INCLUSIVE OR). Given 2 bits as inputs the logical OR output will be 1 if either the first input OR the second input OR both inputs are 1. Stated in logical form



The instruction ORA A #$08 will perform the logical OR operation with ACC A contents and the mask word, 08, as inputs. The result will reside in ACC A. If 144A contains $CA, what will be the result after execution of

```
B6 144A    LDA A $144A
8A 5C      ORA A #$5C
```

Contd...

ACC A/DE

	7	6	5	4	3	2	1	0 ← bit #	
CA =	1	1	0	0	1	0	1	0	
5C	0	1	0	1	1	1	0	0	
	1	1	0	1	1	1	1	0	= DE

The mask word 5C (01011100) with a 1 in bits #2, 3, 4 and 6 ensures that these bits are set, regardless of the original data in address 144A. All other bits remain the same.

2-26

Write the assembly language instruction and machine code to set bits #2 and #7 of the data in address 06A4, without changing the other bits of this data. Use binary format for the mask word.

```
B6 06A4          LDA A  $06A4
8A 84            ORA A  #%10000100
B7 06A4          STA A  $06A4
```

In summary a 1 is used in the mask word of an ORA operation for each bit that is to be set. All other bits in the mask word are 0.

Now set bit #3 and clear bit #5 of address 16D6. Use binary format for the mask words.

```
B6 16D6          LDA A  $16D6
8A 08            ORA A  #%00001000   set bit #3
84 DF            AND A  #%11011111   clear bit #5
B7 16D6          STA A  $16D6
```

X	X	0	X	1	X	X	X

X = unchanged bit

Once more now! Set bits #7, 6 and 2 of address 1A42 and clear bits #1 and 4. Assume that each bit controls the lights for one room in an 8 room house. Provide both assembly language and machine code instructions.

```
B6 1A42          LDA A  $1A42
8A C4            ORA A  #%11000100   (Set 7, 6 and 2)
84 ED            AND A  #%11101101   (Clear 4 and 1)
B7 1A42          STA A  $1A42
```

Although this is the end of the "Accumulator Operations" chapter several other accumulator operations will be introduced at a more appropriate place, later in this workbook. You are probably ready for a change from "bit bashing". Time for a coffee!

SYMBOLIC ADDRESSING

So far we have used absolute addresses e.g., 1A42 for storage of data. When writing in assembly language this is not desirable for several reasons:

- until the program is assembled the addresses available for data storage may not be known.
- if many addresses are used for different purposes it becomes difficult to remember the purpose of each address while preparing the program.
- if a program is later modified certain addresses now used for data storage may not be available, requiring re-assignment of storage addresses.

The solution is the use of a "symbolic address" rather than an absolute address e.g.,

```
STA A COUNTR
```

which stores ACC A contents in an address carrying the symbolic address COUNTR. The absolute address will be determined when the instructions are assembled into machine code and printed on the resultant listing. Meanwhile the programmer can continue to use the symbolic address as if it were an absolute address.

To present an everyday analogy one might suggest meeting for lunch at "Dan's Place" (a symbolic address), whereas Dan's Place might be at 1463 Main Street (the absolute address).

Write the assembly language instructions to initialize the symbolic address COUNTR with the hex value 3C.

```
LDA A  #$3C
STA A  COUNTR
```

Symbolic addresses generated by the programmer can be up to 6 characters long, the first character being a letter and all subsequent characters being a letter or a number. It is good practice to choose a symbolic address which describes the function, COUNTR perhaps being a counter to keep track of the number of events that take place when the program is executed. The only illegal symbolic addresses are A, B and X, the first two being previously assigned to accumulators. Single letters for symbolic addresses are almost meaningless and should be avoided.

Write the assembly language instructions to set bit #5 of STATUS, without changing any other bits.

```
LDA A  STATUS
ORA A  #%00100000
STA A  STATUS
```

Only after the above instructions are assembled into machine code will we know the absolute address for STATUS.

When the computer assembles an assembly language program, it needs to know at what address to start, in assigning each byte of machine code to a memory address. The ORG (origin) directive to the assembler, in the example below, designates the starting address, e.g.,

```
ORG     $0200
LDA A   #$3C
STA A   COUNTR
```

This will cause the following address assignments for the resultant machine code, assuming that COUNTR corresponds to address 0243

```
0200/86 }
0201/3C }- LDA A #$3C

0202/B7 }
0203/02 }- STA A COUNTR
0204/43 }
```

To minimize the amount of paper, produced by the assembler, the address printed is for the first byte of each instruction, e.g.,

```
0200 86 3C          LDA A   #$3C
0202 B7 0243        STA A   COUNTR
```

Write the assembly language instructions and machine code to clear bit #4 of STATUS, which corresponds to address 124E. Start the instructions at address 1200. Show the addresses.

```
                    ORG     $1200
1200 B6 124E        LDA A   STATUS
1203 84 EF          AND A   #%11101111
1205 B7 124E        STA A   STATUS
```

A very common error is omission of the $ symbol, which causes the assembler to interpret 1200 as a decimal number in the above example.

To reserve a memory byte for a specific symbolic address, the assembler MUST be directed to do so. In this program

```
0200                    ORG     $0200
0200 86 3C              LDA A   #$3C
0202 B7 0243            STA A   COUNTR
                        |
                        |
                        |
0243 0001     COUNTR    RMB     1
```

The last line, COUNTR RMB 1 (Reserve Memory Byte - 1 only) causes one byte (address 0243) to be reserved and recognized as the symbolic address COUNTR.

This symbolic address, COUNTR, contains data and must not be embedded in the middle of a group of instructions where its contents would be interpreted as an instruction, rather than data. Such an error is seen in this example:

```
0200                    ORG     $0200
0200 86 4F              LDA A   #$4F
0202 B7 0205            STA A   COUNTR
0205          COUNTR    RMB     1
```

Here COUNTR (address 0205) contains 4F after the first two instructions are executed. The next instruction would then be from the next address, 0205, whose contents is now 4F, a CLR A instruction. It is the execution of the program which determines whether the contents of a memory address is treated as an instruction or data.

To avoid the above problems the symbolic address COUNTR is located outside the group of instructions forming this part of the program, as in the first example.

No answer is required in this frame

Write the instructions to initialize DATA5 with the value A4. Start this program at address 0400 and show a complete listing, noting that DATA5 corresponds to address 0462.

```
0400                    ORG     $0400
0400 86 A4              LDA A   #$A4
0402 B7 0462            STA A   DATA5

 0462 0001    DATA5     RMB     1
```

Label Field | Operator Field | Operand Field | Comment Field

The 4 fields of an assembly language program are seen above. The operator and operand have been discussed previously. In the bottom line we see DATA5, a "label", that is a "symbolic address in the label field". In preparing assembly language programs, labels start in the first column of the line, while operators (LDA etc.) start in the 8th column. It is only necessary to space over 1 column rather than 7 to start the operator (LDA etc.) since the assembler, on noting the absence of a label, will automatically print the operator in the 8th column. Similarly short labels (less than 6 characters) need only to be terminated by one space; the assembler again will start the operator in the 8th column. A sample source program before assembly is shown below.

indented one space to start in the Operator Field.

for Label Field start first column

```
NAM GENPRO
OPT O,S
ORG $1200
LDA A STATUS
AND A #%11101111
STA A STATUS
ORG $124E
STATUS FCB $FF
END
```

It is legal to have more than one ORG directive within a program.

1st column

The comment field, mentioned on the previous page, permits entry of comments to improve the readability of a program, e.g.,

```
LDA A  #$20      INITIALIZE NUVALU
STA A  NUVALU    WITH  20 (DECIMAL 32)
                 ~~~~~~~~~~~~~~~~~~~~~
                      Comment Field
```

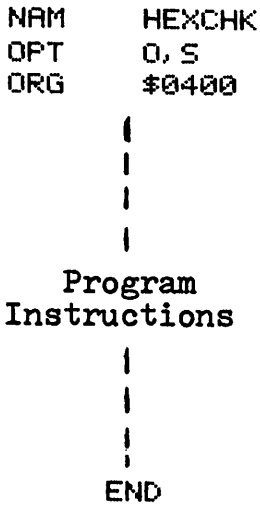
Such comments are ignored by the assembler but printed on the resultant listing. One space is all that is needed to separate such a comment from the operand field.

A good program should begin with a brief description of its purpose and perhaps some of its internal details. Whole lines of comments are legal if the * symbol appears in column 1 of each comment line. These too are ignored by the assembler but printed on the listing. Both examples are seen below.

```
*
* PROGRAM TO OUTPUT TEN CHARACTERS
* TO THE LINE PRINTER.
* VERSION 3B  77/11/12  RWS
*
       LDA A  #$0A      INITIALIZE COUNTER
       STA A  COUNTR    WITH 0A (10 DECIMAL)
```

One assumption to make when programming is that someone else without your help will have to modify your program several years from now. For this, documentation in the form of good comments is essential. To put it more bluntly, if it is not worth documenting it is not worth doing. There will be lots of opportunity to practice this in the next chapter. No answer is required in this frame.

Three other directives are needed to form a complete program. These plus the ORG directive are illustrated below.



The entry following NAM, up to 6 characters long, is a program name, generated by the programmer. It will be reproduced at the top of each page of the assembler's listing, aiding in program recognition.

The OPT (option) directive has many possible entries. The 0, above, requests an object (machine code) file to be produced. Depending on the computer system this file may be stored on paper tape, cassette, diskette or some other medium. The S entry requests a symbol table, a list of all symbolic addresses along with the corresponding absolute addresses, at the end of the listing.

The last directive is END which terminates the assembly language program. Without looking up, try to list the 4 necessary directives for a program.

NAM OPT ORG and END.

To practice use of these directives write a program called CLRALL, starting at address 0400, to clear both accumulators. Yes, it is a ridiculous program.

```
                    NAM     CLRALL
                    OPT     O,S
0400                ORG     $0400
                *
                *CLRALL. . . CLEARS BOTH ACCUMULATORS.
                *
0400 4F             CLR A           A TRIVIAL PROGRAM
0401 5F             CLR B
                    END
```

To save space in this workbook the directives will not normally be shown in the listing, but will be assumed.

Note that END only tells the assembler that this is the end of the program. It does <u>not</u> halt the program, when it is later executed.

In this listing the assembler has noted ERROR 209 for the instruction LDA #$4A. Can you find the error?

```
                    NAM     PROG68
                    OPT     O,S
  0100              ORG     $0100
                *
                * P3-9
                *
  0100 4F           CLR A
  0101 5F           CLR B
ERROR  209
  0102 00 0000  LDA #$4A
  0105 B7 0427      STA A  $0427
                *
                  END
```

The instruction should be LDA A #$4A or LDA B #$4A. Assembler Error Codes, such as ERROR 209, are explained in Appendices J1 and J2.

INDEX REGISTER

Each accumulator is capable of holding 1 byte, represented by 2 hex characters. If 2 bytes are to be referenced we use the Index Register which holds 16 bits (2 bytes or 4 hex characters). The instruction

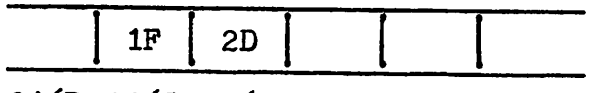
LDX #$1F2D (an IMMEDiate mode instruction)

loads the Index Register with the hex value 1F2D.

The instruction sequence

```
0200 CE 1F2D        LDX    #$1F2D
0203 FF 016C        STX    $016C
```

initializes 2 bytes of memory with 1F and 2D via the Index Register. Address 016C receives 1F while address 016D receives 2D, as shown below.



016B 016C 016D ←memory address

Machine codes for Index Register instructions are on the second page of Appendix C.

Write the instruction sequence to initialize 2 bytes of memory, 14C4 and 14C5, with the hex value 0640. Include the corresponding machine code.

```
0100 CE 0640        LDX    #$0640
0103 FF 14C4        STX    $14C4
```

4-2

Initialize 2 bytes of memory, 1C80 and 1C81, with the hex value 2C40. Include the machine code.

```
CE 2C40        LDX    #$2C40
FF 1C80        STX    $1C80
```

The result is: 1C80/2C (1C80 contains 2C)
1C81/40 (1C81 contains 40)

4-3

A symbolic rather than an absolute address may be used to store the value, e.g.,

```
CE 15D6        LDX    #$15D6
FF 0160        STX    LISTOP
                 |
                 |
0002    LISTOP RMB    2
```

(a) Why does the above example use RMB 2 rather than RMB 1?

(b) Initialize a symbolic address POINTR with the hex value 1C60. Omit machine code this time.

(a) 2 bytes are necessary to store the 2 byte value 15D6.

(b)

```
        LDX    #$1C60 (an IMMED instruction)
        STX    POINTR (an EXTND instruction)
          |
          |
POINTR RMB    2
```

1C goes into POINTR

60 goes into the next address above POINTR.

4-4

The instruction STX POINTR+1 stores the contents of the Index Register in the next address above POINTR. Write an instruction to store the Index Register contents in memory, 3 addresses below CONREG.

FF 14A2 STX CONREG-3

If CONREG correponds to address 14A5, the Index Register contents are stored in address 14A5 - 3 = 14A2, as is seen in the machine code of this listing.

This could be accomplished, one byte at a time, via accumulator operations; however the above approach is preferred because of its simplicity.

4-5

Another use of the index register is seen in

```
LDX    #MESSAG
STX    POINTR
```

which stores the address, not the contents of MESSAG in the 2 byte address, headed by POINTR. If MESSAG corresponds to address 1B34, what will be the contents of POINTR after execution of:

```
LDX    #MESSAG-1
STX    POINTR
```

Write the machine code for these two instructions assuming POINTR corresponds to address 1B6A.

1B33 Since MESSAG corresponds to address 1B34, then MESSAG-1 corresponds to address 1B33.

```
0200 CE 1B33        LDX    #MESSAG-1   <-- IMMED MODE (USES #)
0203 FF 1B6A        STX    POINTR
```

4-6

If TOPBLK corresponds to address 1A00 and contains 03 while TOPBLK+1 contains 80, what is the 2 byte contents of MEMPNT (and MEMPNT+1) for each example below?

```
        LDX     #TOPBLK-1
        STX     MEMPNT
        |
        |
MEMPNT  RMB     2
```

```
        LDX     TOPBLK
        STX     MEMPNT
        |
        |
MEMPNT  RMB     2
```

19FF

1A00 -1 = 19FF, one address below 1A00, now stored in MEMPNT and MEMPNT+1.

0380

The 2 byte contents of TOPBLK and TOPBLK+1 is 0380, now stored in MEMPNT and MEMPNT+1.

4-7

The instruction

CLR 3,X

is interpreted as "Calculate a new address which is the sum of the Index Register contents and the offset, 3 in this example, then clear that memory address." The above instruction could be written as

CLR $3,X

although the $ is redundant for values of 7 or less.

If the Index Register contains 13E4, what address has its contents cleared by CLR 3,X?

13E7 X / 13E4

\+ 3

13E7 = address operated upon by CLR 3,X

This mode of instruction is known as Index Mode. The instruction CLR X is also an Index Mode instruction, being a legal contraction of CLR 0,X. If X contains 2400, the instruction CLR X will clear the contents of address 2400. Similarly LDA A X is a contraction of LDA A 0,X loading ACC A with the contents of the address now in X.

4-8

Write the assembly language instruction to store the contents of ACC A in address 24C0 when the Index Register contains 24A0.

STA A $20,X

```
  24C0
 -24A0
    20
```

Offsets are positive only, 00 to FF, the offset FF producing a new address 255_{10} above the address contained in X. Symbolic offsets, e.g.,

LDA A OFFSET,X

are valid, the value of OFFSET being determined at assembly time. If OFFSET equals $14 via the assembler directive

OFFSET EQU $14

the result would be the same as execution of LDA A $14,X. Assembler directives are normally located at the top of a program, to improve readability

4-9

Machine code for Index mode instructions are found under the INDEX column in Appendix C. Note that

LDA A 3,X (op code A6)

requires 2 bytes as seen by the 2 under the # column, 2 columns to the right of A6. What does the second byte denote? Take a guess. Attempt to encode the above instruction in machine code.

The second byte contains the offset value, 03 in this case, e.g., A6 03

LDA A offset

(Index Mode)

The 2 byte contents of the Index Register can be incremented (1 is added to it) via the instruction

INX - INcrement indeX register (08)

Similarly, DEX - DEcrement indeX register (09)

will decrement it.

Write the assembly language instructions to increment the contents of MEMPNT which now contains the hex value 19FF. What will its new contents (2 bytes) be after the above incrementing?

```
        LDX     MEMPNT
        INX
        STX     MEMPNT
        |
        |
        |
MEMPNT  RMB     2
```

(If not already present in the rest of the program.)

This 3 line sequence will be used many times in this workbook to increment a 2 byte value in memory. Note that the Index Register (X) still contains the incremented value, 1A00 in the above example, after STX MEMPNT is executed.

Another application of Index Mode is seen in code conversion, such as ASCII to Baudot, where each ASCII value is separated in memory from its Baudot value by 80_{16} addresses. Once the address of the ASCII value is known, the corresponding Baudot value is obtained by the instruction LDA A $80,X

To store a message such as "START CARD READER" in memory, it is not necessary to load and store each ASCII character of the message. The sequence below will store each required ASCII code and terminate the message with a null (00).

```
MESSAG FCC    /START CARD READER/
       FCB    0
```

FCC (Form Constant Character) is a directive to the assembler, ordering the storing of the appropriate ASCII codes. Two identical characters are required to define the boundaries of the message. The slash (/) is popular for this since it is not usually used within a message.

FCB (Form Constant Byte) directs the storage of a hex value, 00 in this example, to denote the end of the message. Note the difference between null (00) and the ASCII code for zero (30).

Such message entries generate a lot of unnecessary printing at assembly time as each ASCII character of the message is listed. The OPT directive NOG (NO Generate) eliminates the ASCII code listings but includes the printed message, e.g., OPT O,S,NOG (at the top of the program).

Noting the above message, intialize POINTR with the address one below the start of the message.

```
       LDX    #MESSAG-1
       STX    POINTR
       :
       :
POINTR RMB    2
```

Store the message "ENTER DATA" in memory headed by the label MESS04, and terminated by a null. Initialize MESPNT with the address one below the start of this message.

```
        LDX     #MESS04-1
        STX     MESPNT
        |
        |
        |
MESPNT  RMB     2
MESS04  FCC     /ENTER DATA/
        FCB     0
```

One other assembler directive, available but not required above is FDB (Form Double Byte) e.g.,

FDB $1433,$7

which in this case stores 14 and 33 in 2 bytes, then 00 and 07 in the next 2 bytes. This directive stores an open ended string of 4 character data, each separated by a comma.

What will be the contents of ACC A after execution of the instructions shown below?

```
        LDX     #MESS04-1  INITIALIZE POINTER WITH
        STX     POINTR     ADDRESS MESS04-1
        LDX     POINTR
        INX
        STX     POINTR
        LDA A   X          GET CHAR VIA X
        |
        |
POINTR  RMB     2
MESS04  FCC     /ENTER DATA/
        FCB     0
```

Contd...

45, the ASCII code for E in the message ENTER DATA. POINTR initially contains the address MESS04-1. After the second STX POINTR is executed, both POINTR and X contain the address corresponding to MESS04. Hence E (ASCII code 45) is the first data retrieved via LDA A X.

The above sequence, with additions, will be used many times in this workbook. The advantage of starting with MESS04-1 rather than MESS04 is that X points to the start of the message when LDA A X is executed the first time.

4-14

If address 12A6 contains C4 (12A6 / C4) the instruction

```
LDA A $12A6
```

loads ACC A with C4, the contents of address 12A6.

If address 14A5 and the next address contain 12A6 (14A5 / 12 and 14A6 / A6) then

```
LDX     $14A5     X/12A6
LDA A   X         A/C4
```

also places C4 in ACC A, this time via an "indirect" manner, with X containing the address of the data, 12A6, after execution of LDX $14A5. Hence this is commonly known as an "indirect" or "deferred" memory reference.

This process can be extended further. Given the following initial conditions:

1C50 / 14A5
14A5 / 12A6
12A6 / C4

the instructions

```
LDX   $1C50
LDX   X
LDA A X
```

will also place C4 in ACC A via a "double deferred" memory reference. Before execution of LDX X, X contains 14A5. This instruction, LDX X, loads X with the contents of the address now in X, that is with 12A6 the contents of 14A5. The last instruction then loads C4, the contents of 12A6, into ACC A.

4-15

The main point of this chapter probably needs review again. If X / 13C4 where is the data stored when STA A X is executed?

in address 13C4. The best way to interpret this instruction is "store the data in Accumulator A via X", that is X points to the destination .

4-16

If X / 02AE and 02AE / B5 what will ACC B contain after the instruction LDA B X is executed?

B5 Accumulator B is loaded via X, that is from the address now in X. This time X points to the source of the data.

4-17

If X / 267E what is compared when the instruction CMP A X is executed?

The contents of Accumulator A is compared with the contents of address 267E.

BRANCHING - ASSEMBLY LANGUAGE

Computer programs in which instructions are executed in a simple linear manner are almost non-existent. In fact many decisions are made by computers, in executing a typical program, to determine what to do next. A program with decisions in it is described as follows.

The computer may be required to determine if the ASCII code, now in ACC A corresponds to a valid hex character, e.g., 30 to 39 for 0 to 9 or 41 to 46 for A to F. Invalid characters are to be rejected. Valid ASCII codes are to be converted to their corresponding hex value, e.g., 39 becomes 9 or 46 becomes 0F.

In eliminating invalid ASCII codes the computer must first eliminate all values below 30. The instructions

```
CMP A #$2F    (CoMPare acc A to 2F)
BLS   BADHEX (Branch if Lower or Same to BADHEX)
```

will do this. If the value in ACC A is lower than 2F or the same as 2F, the program will branch to BADHEX; that is the next instruction executed will be the one carrying the label BADHEX.

If the value in ACC A is 30, the ASCII code for 0, what will happen after execution of the above 2 instructions? Take a guess if necessary.

No branching will take place. The next instruction executed will be the one following BLS BADHEX.

If the first test was passed (no branch since the ASCII value was 30 or greater), the next test is to check for values greater than 39, the ASCII code for 9. If the value is 39 or lower, the program should branch to NUMOK, otherwise it should continue. Write the instructions to do this noting the availability of the instructions:

BLS - Branch if Lower than or Same
BHI - Branch if HIgher than
BRA - BRAnch unconditionally.

```
CMP A  #$39
BLS    NUMOK     0 TO 9. VALID HEX
```

The conditional branch instructions BLS and BHI treat the ACC A contents as an unsigned number, that is all values, 00 to FF are considered positive.

By having available both BLS and BHI (opposite instructions) the programmer can either choose to branch or not to branch when a specific condition is met.

So far the program is:

```
HEXCHK CMP A  #$2F
       BLS    BADHEX    MUST BE BELOW 30
       CMP A  #$39
       BLS    NUMOK     MUST BE 30-39
        :
        :
NUMOK   |
        :
BADHEX
```

For ASCII codes 30 - 39 we want the hex values 0 - 9 in ACC A. What instruction, starting at the label NUMOK will *do* this, e.g., when key 5 on a keyboard is struck the final contents of ACC A will be 5, not 35. The program should go to GOODHX when the correct value is in ACC A. Again assume that the ASCII code is already in ACC A when the program starts. Show only the program additions.

```
NUMOK   SUB A   #$30
        BRA     GOODHX
```

or

```
NUMOK   SUB A   #'0
        BRA     GOODHX
```

We now have:

```
HEXCHK  CMP A   #$2F
        BLS     BADHEX    MUST BE BELOW 30
        CMP A   #$39
        BLS     NUMOK     MUST BE 30-39
        |
        |
GOODHX  |
        |
        |
NUMOK   SUB A   #$30
        BRA     GOODHX
BADHEX
```

Now screen for values A to F. Valid characters in this group should be converted from their ASCII code to their true hex value, e.g., 0A when A is struck. For valid characters continue to GOODHX, the next line, after this conversion. For invalid characters branch to BADHEX.

```
        CMP A   #$40
        BLS     BADHEX    MUST BE 3A-40
        CMP A   #$46
        BHI     BADHEX    MUST BE GREATER THAN 46
        SUB A   #$37      41-46 NOW 0A-0F
GOODHX                    END OF ROUTINE.
```

The ASCII code for A is 41, for which the hex value is 0A. The difference is 37, which when subtracted from 41 gives us 0A. Similarly when F is struck, 46 - 37 = 0F. Calculations are shown below:

```
 FF          41
-0A         +F6
 F5          1 37
+ 1
 F6 = -0A
```

```
When A is struck   41   ASCII for "A"
                  +C9   (-37)
                  1 0A
                     <- hex code for A
```

The final version of this routine (let's call it HEXCHK) is:

```
HEXCHK... CHECKS IF CHAR NOW IN ACC A
IS VALID HEX CHAR. THAT IS 0-9 OR A-F.
ENTER WITH ASCII CHAR IN ACC A.
RETURNS WITH 4 BIT EQUIVALENT HEX IN ACC A IF VALID


HEXCHK CMP A  #$2F
       BLS    BADHEX    MUST BE BELOW 30
       CMP A  #$39
       BLS    NUMOK     MUST BE 30-39
       CMP A  #$40
       BLS    BADHEX    MUST BE 3A-40
       CMP A  #$46
       BHI    BADHEX    MUST BE GREATER THAN 46
       SUB A  #$37      41-46 NOW 0A-0F
GOODHX                  END OF ROUTINE.
       :
       :
       :
NUMOK  SUB A  #$30
       BRA    GOODHX
BADHEX
       :
       :
       :
BADMES FCC    /NOT VALID HEX/
       FCB    0
       END
```

What would happen if the first line was CMP A #$30?

When 0 is struck on the keyboard the ASCII code 30 would result. The first 2 lines would then cause a branch to BADHEX (normally reserved for invalid characters), since BLS BADHEX recognizes that the code produced is the same as 30. Such an error where a branch instruction is incorrect for one value, is very common. Hence a programmer should manually check for boundary values, 0, 9, A and F in the above program.

The label GOODHX could provide an instruction JMP NEXT, jumping to the next program segment. The BADHEX section could be temporarily terminated by the instruction BADHEX BRA BADHEX, an instruction which loops back to itself, preventing execution of "left over" code in that memory address.

5-6

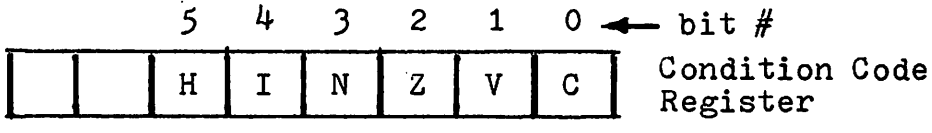
Modify this HEXCHK program to include the necessary assembler directives, this time calling the program HEX2C and starting it at address 1E40. Show only the first and last lines of the program.

```
        NAM     HEX2C
        OPT     O,S
        ORG     $1E40
HEXCHK  CMP A   #$2F
    :
    :
BADHEX
        END
```

Note that all 4 directives appear in the operator field. The first label of the program does not have to agree with that used with NAM. The latter usually designates which version is listed, e.g., version 2C in this example. Updating the version number when changes are made is a very effective way of denoting which listing is the latest, an absolute essential as programs evolve.

5-7

To understand better how the branch instructions operate one must be aware of the Condition Code Register (CCR) in which each of the 6 assigned bits may be set or cleared according to each instruction executed.



For example bit #0 is the CARRY or C bit which will be set if an 8 bit addition produces an overflow, the C bit behaving as the 9th bit. The C bit can be set under other conditions, seen later.

Bit #1, the oVerflow or V bit, is set if a 2's complement (signed number) arithmetic operation produces an answer exceeding the range of -128_{10} (80_{16}) to $+127_{10}$ ($7F_{16}$), the available range using an 8 bit signed number.

Contd...

The Z or Zero bit (bit #2) is set when a zero is produced in a memory or accumulator operation, e.g., CLR A or CLR MEMPNT.

The N or Negative bit (#3) is set when a resultant leading bit = 1, implying a negative value in the accumulator or memory.

The I bit will be treated in the Interrupt chapter.

The H bit is used internally by the DAA instruction for BCD arithmetic operations. (Details in Appendix K)

Each instruction executed affects the CCR bits as noted in the right column of Appendix C where the state of each CCR bit, after the execution of each instruction, is shown. For example, CLR A will clear or reset (R) the N, V and C bits and set (S) the Z bit. The dot implies no change. The vertical arrows for the CMP instruction imply conditional setting or clearing of these bits. For example, CMP A #$72 produces a subtraction (ACC A minus 72) which sets the Z bit if the result is zero or sets the N bit if the answer is negative and/or sets the V bit if a two's complement overflow took place.

Detection of the Z bit status is achieved via

BEQ - Branch if EQual (Equal to Zero if no other reference named)

or BNE - Branch if Not Equal

as seen in

```
DEC A
BEQ   ALLDUN
```

which branches to ALLDUN if ACC A = 0. Similarly BNE branches on non-zero results when

```
LDA A  SUBTOT
AND A  #$C2
BNE    MATCH
```

is executed. Will branching occur assuming SUBTOT/3E? What is the Z bit state,

Yes branching will occur since C2·3E = 2 (not equal to zero), clearing the Z bit and causing a branch via BNE MATCH.

5-8

Will the following instructions cause a branch to HIT if KEDATA contains 29?

```
LDA A  KEDATA
AND A  #$D6
BNE    HIT
```

NO

KEDATA = 00101001
D6 = 11010110
LOGICAL AND = 00000000

Since the result is zero the BNE instruction (Branch if <u>not</u> equal to zero) will not cause a branch to HIT. The Z bit will be set.

5-9

The instructions:

```
LDA A  CONTRO
BIT A  #$40
BNE    HIBIT
```

perform the logical AND on CONTRO and 40, without modifying ACC A. The CCR bits are affected and branching to HIBIT will occur if bit #6 of CONTRO = 1 (not equal to zero).

XXXXXXXX CONTRO
01000000 40
↑
Bit #6 is only bit of CONTRO tested.

Since the BIT instruction does not destroy the original contents of ACC A, several bits can be individually tested, permitting multiple branches.

Write the instructions to branch to RECEIV if bit #0 of SERCSR is set or to TRANS if bit #1 of SERCSR is set; otherwise continue.

Contd...

5-9
Contd.

```
LDA A   SERCSR
BIT A   #$01
BNE     RECEIV
BIT A   #$02
BNE     TRANS
|
|
|
|
```

5-10

Write the instructions to test bits #2 and 3 of SPEED, branching to LSPEED if bit #2 is set, to HSPEED if bit #3 is set or to STOPIT if both bits are cleared. Assume that both bits will not be set at the same time.

```
LDA A   SPEED
BIT A   #%00001100   CHECK FOR 00
BEQ     STOPIT
BIT A   #%00000100   CHECK FOR BIT #2=1
BNE     LSPEED
BIT A   #%00001000   CHECK FOR BIT #3=1
BNE     HSPEED
|
|
|
|
|
```

Note that all bits of ACC A, "viewed" via the mask word, must be zero to set the Z bit of the CCR. Hence both bits #2 and #3 of SPEED must be zero to branch to STOPIT via the above test. The above instructions could be part of a speed control routine for a machine, the individual bits of SPEED being controlled by the machine's push buttons, connected to the computer.

Further branching operations will be seen in a program to clear a group of memory locations. In the program below, what is the initial contents of MEMADD? What address will be first to be cleared?

```
0200 CE 23FF          LDX    #$2400-1
0203 FF 0260          STX    MEMADD
0206 FE 0260 MORCLR   LDX    MEMADD
0209 08               INX
020A FF 0260          STX    MEMADD
020D 6F 00            CLR    X
020F 20 F5            BRA    MORCLR

0260                  ORG    $0260
0260 0002     MEMADD  RMB    2
```

Initially MEMADD contains 23FF (2400 - 1 = 23FF).
INX will increment X to 2400, the first address to be cleared via CLR X.

What address will be cleared when CLR X is executed the second time? Explain, starting at MORCLR (second time through here). When does this clearing operation cease?

Address 2401
When MORCLR LDX MEMADD is executed the second time X contains 2400. After INX, X contains 2401 which is stored via STX MEMADD. CLR X then clears address 2401.

This clearing operation will continue until the above program is partially overwritten (cleared) by its own operation. We need a method to break out of this loop after a specific address is cleared. If the suspense is killing you, check the next page!

The CPX (ComPare indeX register) instruction compares the Index Register contents to some 2 byte reference value, e.g.,

CPX #$24C7

or CPX HIVALU

Only 2 branch instructions are valid after CPX, BEQ or BNE.

Modify the previous program to exit from the loop after address 240F is cleared.

```
0200 CE 23FF          LDX     #$2400-1 THIS PROGRAM CLEARS
0203 FF 0260          STX     MEMADD   AND LOOPS BACK
0206 FE 0260 MORCLR   LDX     MEMADD   UNTIL MEMORY ADDRESS
0209 08               INX              240F IS CLEARED
020A FF 0260          STX     MEMADD   AFTER WHICH EXIT
020D 6F 00            CLR     X        TAKES PLACE
020F 8C 240F          CPX     #$240F
0212 26 F2            BNE     MORCLR
                 *
0260                  ORG     $0260
0260 0002     MEMADD  RMB     2
                      END
```

While it is true that the Index Register could remain the pointer throughout this program, without using MEMADD, we are looking ahead to programs where the Index Register is used for several purposes inside one loop, requiring retrieval and storage of each memory address pointer each time it is used.

How many memory locations will be cleared by the previous program?

10_{16} or 16_{10}

After CLR X is executed	X/	#of addresses cleared
1st time	2400	1
2nd time	2401	2
3rd time	2402	3
'	'	'
'	'	'
'	'	'
15th time	240E	$0F_{16}$ (15_{10})
16th time	240F	10_{16} (16_{10})

Tables like this are useful to ensure that the exit from a loop takes place at the correct point, not one loop too soon or late. For example, if the problem was to clear 20_{16} locations such a table ensures that 241F is the correct reference address for the exit.

Modify the previous program to clear 100_{10} memory addresses, starting at address 2400. Show only the changes.

CPX #$2463 is the only change.

$100_{10} = 64_{16}$

Memory Address	# of addresses cleared
2400	1
2401	2
'	'
'	'
'	'
2462	63_{16} (99_{10})
2463	64_{16} (100_{10})

What would be the effect if the label MORCLR appeared opposite the first instruction, e.g.,

MORCLR LDX #$2400-1

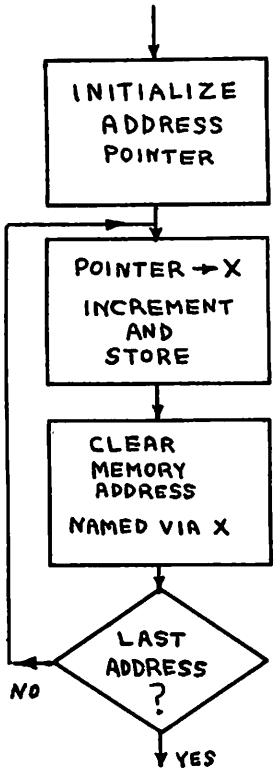
rather than in its present location? Refer back several frames for the program.

The program would be re-initialized after each loop, hence it would clear address 2400 each time in a continuous loop. This is a fundamental error which everybody makes at least once, including you and me. The only question is when. More important though is to be aware of this potential problem. The solution can be summarized by

LOOPBACK IS ALWAYS BELOW INITIALIZATION

Initialization in the previous program sets up MEMADD with 23FF, its initial value. The program loops back to MORCLR, below the initialization in the original program.

Good programming requires good planning. While many planning methods are advocated today, one of the simplest and most effective is the flow chart, shown below.



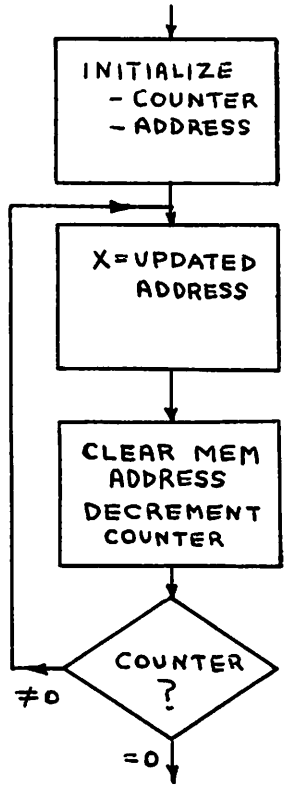
Note that a flow chart depicts functions, not specific instructions.

Here operations such as initialization, clearing, storing, etc., are shown inside rectangles. Decisions are depicted by diamonds which have multiple exits, the chosen path depending on the decision made.

A good flow chart represents the major effort in preparing a program. Converting it to instructions, once you are familiar with the instruction set, should take less time than flow charting. A flow chart is also useful in documenting a program for use by future users.

No answer is required in this frame.

The program to clear 64_{16} locations could be handled by using a counter, with an initial value of 64_{16}, which is decremented after each address is cleared. Exit would then take place when the counter is zero. Flow chart such a program.



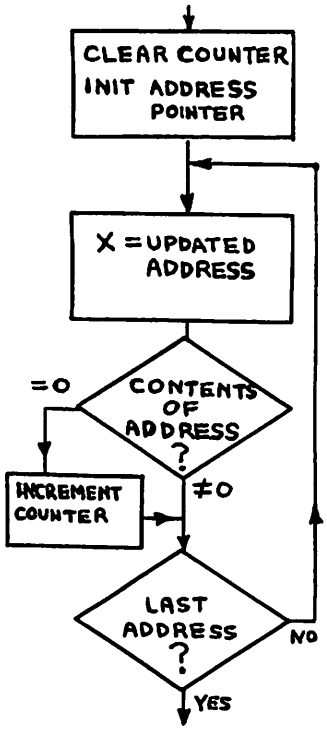
To next part of longer program.

Now write the program to clear 100_{10} (64_{16}) locations, starting at address 1200. The program itself is called MEMCLR and should start at address 0800. Include the necessary assembler directives. The instructions INC or DEC may be useful to you.

```
                        NAM     MEMCLR
                        OPT     O,S
0800                    ORG     $0800

                MEMCLR...CLEARS 100 (DECIMAL)  MEMORY LOCATIONS
                STARTING AT 1200.  USES X.

0800 86 64    MEMCLR LDA A   #$64       OR LDA A #100
0802 B7 0260         STA A   COUNT      INIT COUNTER
0805 CE 11FF         LDX     #$1200-1
0808 FF 0261         STX     MEMADD     SET UP ADDRESS POINTER.
080B FE 0261 MORCLR  LDX     MEMADD
080E 08              INX
080F FF 0261         STX     MEMADD     GET ADDRESS
0812 6F 00           CLR     X          AND CLEAR IT
0814 7A 0260         DEC     COUNT      LAST ADDRESS?
0817 26 F2           BNE     MORCLR     NO.  TRY AGAIN
                     !
0260                 ORG     $0260
0260 0001     COUNT  RMB     1
0261 0002     MEMADD RMB     2
                     END
```

COUNT could have been incremented from 0, exit taking place when count equals 64. Down counting is preferred since it is easier to detect zero than a specific value (CMP A #$64). Both, however, are valid.

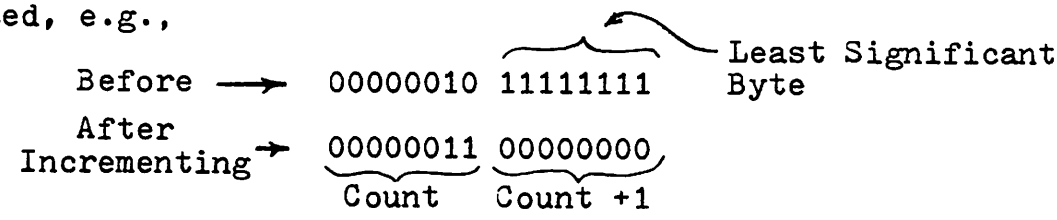
In the previous program the task was to clear an address. In the next program the task is to count the number of addresses, 0900 to 09FF inclusive, which contain zero. This time the task itself will contain a decision, to count or not to count. First flow chart, then write the program.



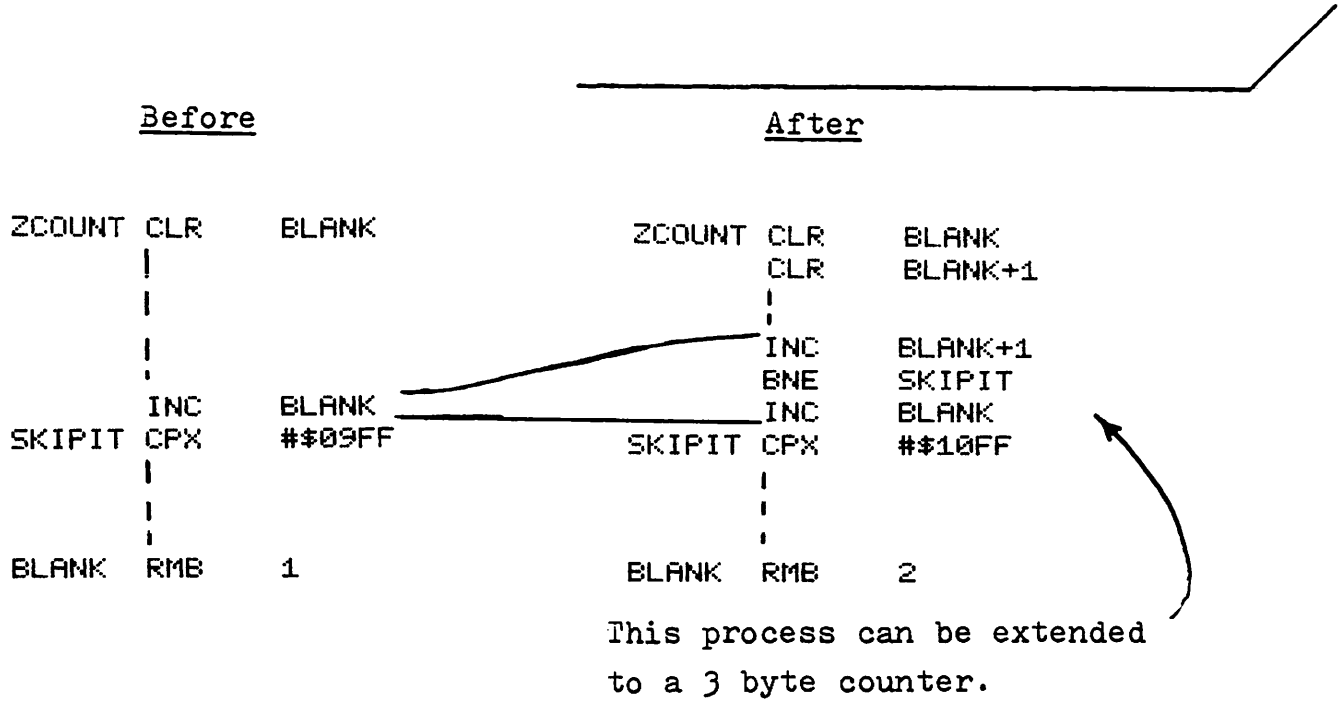

```
0200 7F 0262 ZCOUNT CLR     BLANK     EMPTY COUNTER
0203 CE 08FF        LDX     #$0900-1
0206 FF 0260        STX     MEMPNT    INIT ADDRESS POINTER
0209 FE 0260 MORCHK LDX     MEMPNT
020C 08             INX
020D FF 0260        STX     MEMPNT    GET NEXT ADDRESS
0210 A6 00          LDA A   X         GET ITS CONTENTS
0212 26 03          BNE     SKIPIT    NOT ZERO
0214 7C 0262        INC     BLANK     GOT ONE
0217 8C 09FF SKIPIT CPX     #$09FF    LAST ADDRESS?
021A 26 ED          BNE     MORCHK    NO.  BACK AGAIN
                    !
0260                ORG     $0260
0260 0002    MEMPNT RMB     2
0262 0001    BLANK  RMB     1
                    END
```

ACC B, if available, could have been used as the counter.

When the possible count exceeds 255_{10} (FF_{16}) two bytes will be necessary to contain the number of bits. A problem in incrementing a 16 bit (two byte) counter exists when the low byte overflows to zero, at which point the high byte must be incremented, e.g.,



Modify the previous program to count the number of addresses containing zero in the address range 0900 to 10FF inclusive. Show program changes only.



The Index Register also can be used to increment a 2 byte counter. What changes would you make from the previously modified program to use the Index Register to increment BLANK? Again show only the program changes.

Before

```
        INC     BLANK+1
        BNE     SKIPIT
        INC     BLANK
SKIPIT
```

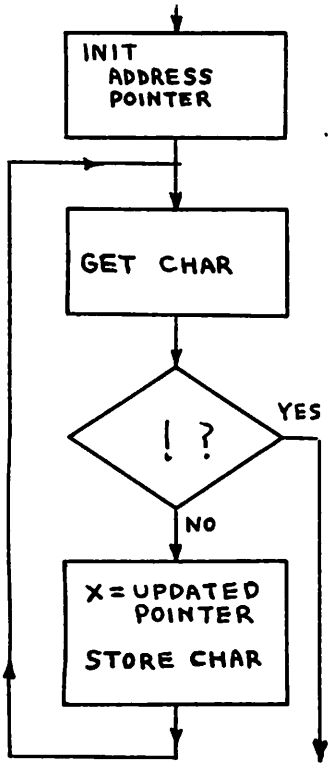
After

```
        LDX     BLANK
        INX
        STX     BLANK
SKIPIT
```

If BLANK is to be tested or compared later, the Index Register will be needed for that operation. Hence the second solution, using the Index Register, is preferred.

The second solution shows how the Index Register can be used for many tasks within a program since the updated value (after INX) is immediately stored in memory, releasing the Index Register for another task.

Assume that the instruction JSR GETCHR, a subroutine call which we'll examine in detail in a later chapter, puts the ASCII code for the key, struck on a keyboard, into ACC A. Use this instruction within a looping type program to store in memory the ASCII codes for the keys struck. Start storing data at address 1200. When the ! key is struck, exit from the loop without storing this terminator character. First flow chart your program.



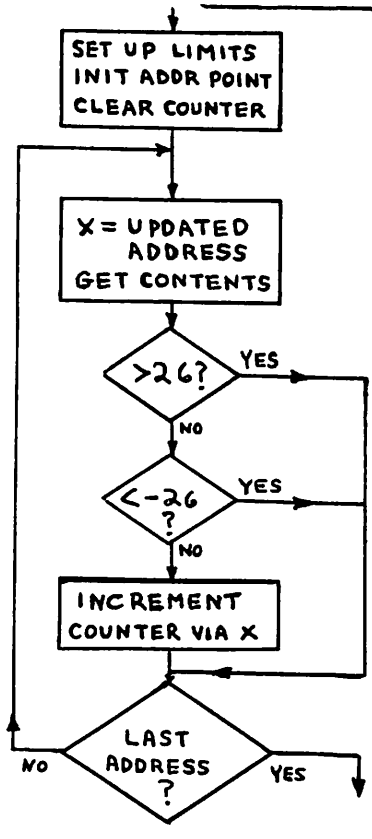

```
GETCHR EQU    $1F00

  STOASC. . STORES ASCII CODES FROM KEYBOARD
  IN SUCCESIVE MEM ADDR STARTING AT 1200.
  ! TERMINATES PROGRAM.
  CALLS GETCHR.  USES A AND X.

STOASC LDX    #$1200-1
       STX    ADDRES    INIT POINTER
GETMOR JSR    GETCHR    GET ASCII CODE
       CMP A  #'!
       BEQ    ALLDUN    MUST BE !
       LDX    ADDRES
       INX
       STX    ADDRES    UPDATE ADDRESS
       STA A  X         AND STORE ASCII CODE
       BRA    GETMOR    AND BACK AGAIN.
ALLDUN

ADDRES RMB    2
```

Here the test takes place before the task, to avoid storing the ! character.

Branching instructions recognizing signed ($\pm$) values are:

BGE - Branch if Greater or Equal
BGT - Branch if Greater Than
BLE - Branch if Less than or Equal
BLT - Branch if Less Than
BPL - Branch if PLus
BMI - Branch if MInus

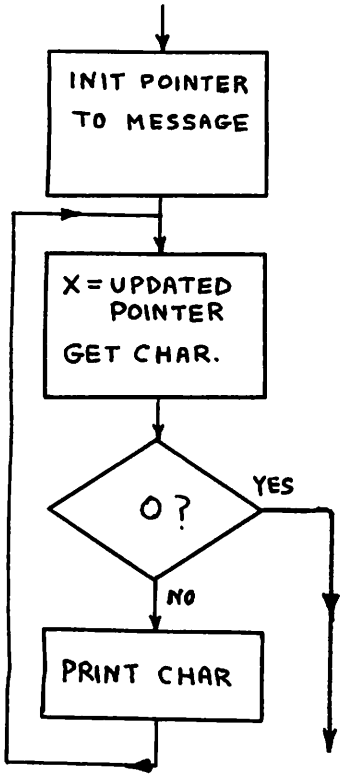
Flow chart a program to count the number of occurrences of values between $\pm$ 26_{16} inclusive, within the memory range 0800 - 0BFF inclusive. Manually check your program for proper branching for values of $\pm$26 and $\pm$27.



From your flow chart on the previous page, write the program.

```
                    MEMCHK. . COUNTS OCCURRENCES OF +26 TO -26 HEX
                    IN MEM ADDR 0800-0BFF INCLUSIVE

0200 86 26   MEMCHK LDA A  #$26
0202 B7 0271        STA A  HILIM     SET UPPER CHECK VALUE
0205 40             NEG A
0206 B7 0270        STA A  LOLIM     SET LOWER CHECK VALUE
0209 7F 0274        CLR    HIT
020C 7F 0275        CLR    HIT+1
020F CE 07FF        LDX    #$0800-1
0212 FF 0272        STX    MEMPNT    INIT POINTER
0215 FE 0272 GETBYT LDX    MEMPNT
0218 08             INX
0219 FF 0272        STX    MEMPNT    GET NEXT ADDRESS
021C A6 00          LDA A  X         GET CHAR
021E B1 0271        CMP A  HILIM     >26?
0221 2F 0C          BGT    NOHIT     IF SO IGNORE IT
0223 B1 0270        CMP A  LOLIM     <26?
0226 2D 07          BLT    NOHIT     IF SO IGNORE IT
0228 FE 0274        LDX    HIT
022B 08             INX
022C FF 0274        STX    HIT       ADD 1 TO HIT
022F FE 0272 NOHIT  LDX    MEMPNT
0232 8C 0BFF        CPX    #$0BFF
0235 26 DE          BNE    GETBYT    NO, BACK AGAIN


0270
0270 0001    LOLIM  RMB    1
0271 0001    HILIM  RMB    1
0272 0002    MEMPNT RMB    2
0274 0002    HIT    RMB    2
```

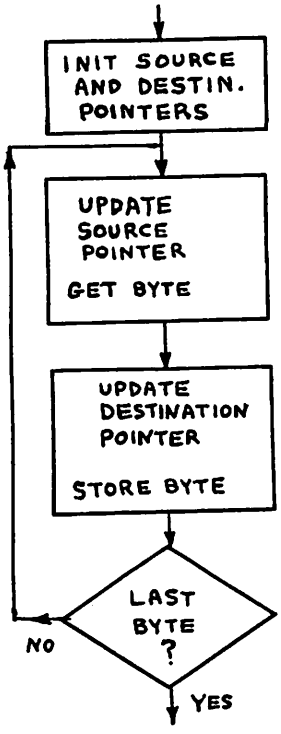
Previously we saw how to store a message in memory. It is time to print such a message. For now, assume that the instruction JSR PRINT, a subroutine call, prints the contents of ACC A as one ASCII character on a printer. Assume that the label MESSAG heads a stored message, in ASCII format, terminated by a null. Flow chart and write a program to print this message, using the JSR PRINT instruction. If you are stuck, look at the first two instructions of the solution




```
   MESSPR. . PRINTS MESSAGE THAT IS STORED IN MEMORY.
   CALLS PRINT SUBROUTINE FOR EACH CHARACTER PRINTED.
   USES A AND X PLUS PRINT SUBROUTINE.

MESSPR LDX    #MESSAG-1
       STX    POINTR    INIT MEM POINTER
MORPRT LDX    POINTR
       INX
       STX    POINTR    GET ADDRESS OF CHAR
       LDA A  X         GET ASCII CHAR IN A.
       BEQ    ALLDUN
       JSR    PRINT     PRINT IT
       BRA    MORPRT    BACK FOR MORE
ALLDUN
       |
POINTR RMB    2
MESSAG FCC    /FILENAME?/
       FCB    0
```

Note the test before printing to avoid trying to type a null which cannot be printed.

Data stored on a diskette, a magnetic mass storage device, is usually written in blocks of 80_{16} characters at a time from a buffer, which is a specific block of memory. In such an operation the X register must be used both for retrieving data from the "source" memory address and for storing it in the "destination" address. For this 2 pointers must be initialized. For each byte moved, each pointer must then be updated for use by X. With this in mind, flow chart and write a program to move the memory block 0600 - 06FF to 0800 - 08FF.




```
MOVEIT  LDX     #$0600-1
        STX     SOURCE    INIT SOURCE ADDRESS
        LDX     #$0800-1
        STX     DEST      INIT DESTINATION ADDRESS
MOVBYT  LDX     SOURCE
        INX
        STX     SOURCE    GET NEXT SOURCE ADDRESS
        LDA A   X         GET A BYTE
        LDX     DEST
        INX
        STX     DEST      GET DESTINATION ADDRESS
        STA A   X         AND STORE BYTE
        CPX     #$08FF    LAST BYTE?
        BNE     MOVBYT    NO. AROUND AGAIN

SOURCE  RMB     2
DEST    RMB     2
        END
```

Earlier we saw how to increment a 2 byte counter without using the X Register. Similarly a 2 byte counter can be decremented without using the X Register. A special condition, shown below, exists when the least significant byte is zero, before decrementing, since both bytes will have to be decremented this time.

	Count	Least Significant Byte
Before Decrementing	00111011	00000000
After Decrementing	00111010	11111111
	Count	Count +1

Write the instructions to decrement the two byte counter COUNT, recognizing the special condition above. The instruction TST (TeST or "compare to zero") is useful here.

```
        TST     COUNT+1  CHECK LEAST SIG BYTE FOR ZERO
        BNE     DECLOW   IF NOT 0 IGNORE MOST SIG BYTE
        DEC     COUNT    IF LEAST SIG BYTE 0 DEC MOST
DECLOW  DEC     COUNT+1  ALWAYS DEC LEAST SIG BYTE
```

This sequence of instructions is most useful if a 2 byte counter must be decremented when the Index Register is not available to do it. This process also can be extended to a 3 byte counter.

The program listed below is a slightly shorter version of HEXCHK, developed earlier in this chapter. This one uses signed branch instructions which had not been discussed when the original program was developed.

```
HEXCHK SUB A  #$30
       BMI    BADHEX   BELOW 30
       CMP A  #$09
       BLE    ENDHEX   BELOW 39.  ABOVE 30
       SUB A  #$07
       CMP A  #$0F
       BHI    BADHEX   46-30-7=0F.  ABOVE F
       CMP A  #$09
       BLE    BADHEX   41-30=0A BELOW A

ENDHEX
BADHEX
```

Since either 30 or 37 had to be subtracted to convert to hex, 30 was subtracted immediately. Branching on a minus value is now possible, eliminating a CMP instruction. While the purpose of this workbook is to help you learn fundamentals rather than write "tight" programs, the above listing is included to point out that the shortest programs are not necessarily the most readable and vice versa.

Time for a break. This was a long chapter.

BRANCHING - MACHINE CODE

Even when writing very short machine code programs it is highly desirable to start with assembly language instructions and then assemble them into machine code. Manual assembly of a program raises a problem in that the address for MEMADD in the instruction STX MEMADD is often not known until MEMADD RMB 2 is encountered, perhaps many instructions later. The solution proposed is the one used by the computer when it assembles a program, that of processing the assembly language program twice. When the assembly language program is read the first time, an absolute address is assigned to each label (symbolic address in label field). During the second reading, machine code is produced for each instruction.

To assign absolute addresses to labels requires knowing how many bytes each instruction requires. This data is available in Appendices C1 and C2, under the # column, for each mode available. Assuming Extended Mode for the instruction LDX MEMPNT, we see 3 in the # column for the "EXTND" mode opposite the LDX instruction.

For the program below assign the appropriate addresses, starting at 0618. Addresses already are assigned to the first 2 instructions.

```
0618          INIT    LDA A   #$17
061A                  STA A   ENDVAL
                      LDX     #$06D7
                      STX     MEMADD
                      RTS
              ENDVAL  RMB     1
              MEMADD  RMB     2
```

```
0618          INIT    LDA A   #$17
061A                  STA A   ENDVAL
061D                  LDX     #$06D7
0620                  STX     MEMADD
0623                  RTS
0624          ENDVAL  RMB     1
0625          MEMADD  RMB     2
```

Now that all addresses are known, complete the assembly operation by assigning the machine code for each instruction. No entry is required for the labels ENDVAL and MEMADD at the end of this program.

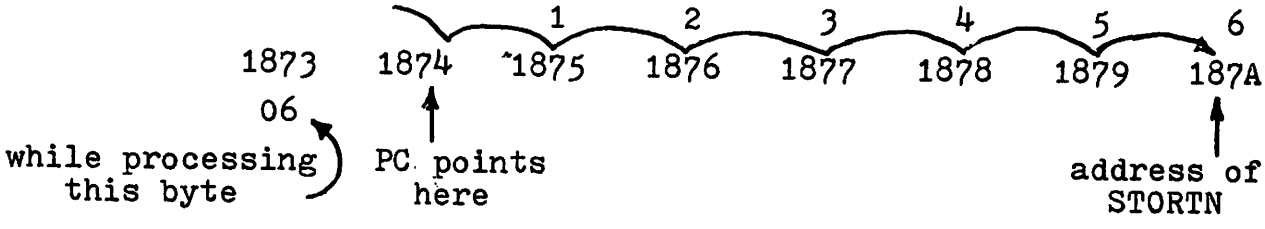
```
0618          INIT    LDA A   #$17
061A                  STA A   ENDVAL
061D                  LDX     #$06D7
0620                  STX     MEMADD
0623                  RTS
0624          ENDVAL  RMB     1
0625          MEMADD  RMB     2
```

```
0618 86 17    INIT    LDA A   #$17
              *
061A B7 0624          STA A   ENDVAL
              *
061D CE 06D7          LDX     #$06D7
              *
0620 FF 0625          STX     MEMADD
              *
0623 39               RTS
              *
0624 0001     ENDVAL  RMB     1
              *
0625 0002     MEMADD  RMB     2
```

In general it is easy to work with the machine code for the 6800 microcomputer. Only one area, that of encoding branch instructions, requires extra care. In the instruction sequence:

```
186F 8C 1A7F         CPX    #START
1872 26 06           BNE    STORTN
1874 CE 187B         LDX    #BIGSOR
1877 BD 1F0C         JSR    OUTMES
187A 39       STORTN RTS
```

the code for BNE is 26. The next byte, 06, is a forward reference to STORTN, 6 bytes beyond the byte following 06. Better read that again! When the microprocessor has fetched 06 from memory and is processing it, to determine the address to which to branch, the program counter (PC) contains the address of the next byte, 1874. It is 6 bytes (hence the 06) from 1874, the PC contents, to 187A, the address of STORTN.



If STORTN is at address 187E instead of 187A, while the BNE instruction remains at the same address, what value is in address 1873, the forward reference to STORTN for the BNE instruction?

0A

187E - 1874 = 0A ← branch offset

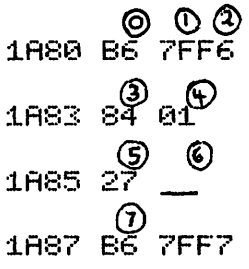
target address — address following branch offset

Backward branching is somewhat more challenging, e.g.,

```
1A80 B6 7FF6 MORTES LDA A  SERCSR
1A83 84 01          AND A  #$01
1A85 27 F9          BEQ    MORTES
1A87 B6 7FF7        LDA A  SERBUF
```

While processing the branch offset F9 (address 1A86) the PC contains 1A87, the address of the next byte. The target address is 1A80, 7 bytes backward from the PC value. Hence F9 (-7) is the branch offset.

To determine this value, F9, the most direct method is to calculate 1A80 - 1A87 resulting in FFF9 as a 2 byte negative value which contracts to F9 as a one byte negative value (refer to the first chapter for 2 versus 1 byte negative numbers). For short backward branches the number of bytes can be determined by counting from 1A80 to 1A87, e.g.,



Since the separation is 7 bytes then -7 can be converted to F9. The missing value above then becomes F9. For more than a dozen bytes this may become tedious. For short branches, however, it is simple and quick.

No answer is required in this frame.

With more experience in using machine code, you may prefer to count the number of bytes backwards instead of forward to obtain the branch offset directly. Using the previous program this would be:

```
      00 FF FE
1A80 B6 7FF6
      FD FC
1A83 84 01
      FB FA
1A85 27 __  <-------------- F9 then follows the 27
      F9
1A87 B6
```

Using the above technique determine the machine code for the backward branch below. The address for LOOPNO is 1A60.

```
0200 7A 1A60 NOTYET DEC    LOOPNO
0203 27 __          BEQ    NOTYET
0205 4F             CLR A
```

```
      00 FF FF
0200 7A 1A60 NOTYET DEC    LOOPNO
      FD FC     *
0203 27 FB          BEQ    NOTYET
      FB        *
0205 4F             CLR A
                  *
```

```
NEXCHR JSR    GETCHR
       LDX    MEMADD
       INX
       STX    MEMADD
       LDA A  X
       CMP A  #$0D
       BEQ    ENDLIN
       JSR    OUTERM
       BRA    NEXCHR
ENDLIN RTS
*
MEMADD RMB    2
```

Manually assemble the program (opposite) using both the first and last methods to determine each branch offset. Machine code for JSR GETCHR is BD 1F00 and for JSR OUTERM is BD 1F03. Start at address 0740.

```
00 FF FE   0740 BD 1F00 NEXCHR JSR    GETCHR
                        *
FD FC FB   0743 FE 0756        LDX    MEMADD
                        *
FA         0746 08             INX
                        *
F9 F8 F7   0747 FF 0756        STX    MEMADD
                        *
F6 F5      074A A6 00          LDA A  X
                        *
F4 F3      074C 81 0D          CMP A  #$0D
                        *
F2 F1      074E 27 05          BEQ    ENDLIN
                        *
F0 EF EE   0750 BD 1F03        JSR    OUTERM
                        *
ED EC      0753 20 EB          BRA    NEXCHR
                        *
EB         0755 39      ENDLIN RTS
                        *
           0756 0002    MEMADD RMB    2
                               END
```

(0) (1) (2) (3) (4) (5)

Backward branch counting (last method)

address

machine code

0755
-0750
5

(forward reference)

0740 FFFF
-0755 → -0755
=F8AA
+ 1
F8AB
+0740
FFEB

but FFEB (in 2 byte format) becomes EB in 1 byte format (see Chapter 1). Normally JMP NEXCHR rather than BRA NEXCHR would be used to avoid offset calculations.

Branch instructions use a one byte signed offset, limiting the branching range to ±127 (decimal) addresses. Attempted branches beyond this range produce an error at assembly time. Sometimes programs which were previously error-free now will cause a branching error when new instructions, inserted between the branch instruction and the target address, now produce too great an offset. One solution is to branch to the end of the present routine, or some other appropriate place where a JMP (JuMP) instruction, which can jump anywhere, jumps to the target address.

Such a solution is also one way to avoid backward branching in machine code, a pragmatic if not aesthetic solution. Similarly BSR should be replaced by JSR when writing in machine code unless memory locations are scarce.

Assume that NUCHAR, at address 0608 is beyond branching range of BEQ NUCHAR, below. Modify the program to reach NUCHAR. Show your changes in machine code.

```
0200 81 0A          CMP A  #$0A
                 *
0202 27             BEQ    NUCHAR
                 *
0204 BD 1A64        JSR    STORE
                 *
0207 39             RTS
```

```
0200 81 0A          CMP A  #$0A
                 *
0202 27 04          BEQ    JUMPNU
                 *
0204 BD 1A64        JSR    STORE
                 *
0207 39             RTS
                 *
0208 7E 0608 JUMPNU JMP    NUCHAR
```

Changed lines are circled.

A problem often encountered in writing machine code programs is the need to insert a few instructions in the middle of a program. This results in new addresses for all labels below the insert (on the listing) requiring re-encoding of the program.

To prevent or minimize such problems it is desirable to leave memory address gaps between subroutines or program segments, typically 1/4 the length of the code written. Where instructions follow one another continuously for more than ten lines, insert several NOP (No OPeration) instructions (OP CODE 01) which do absolutely nothing except to occupy memory locations. These are easily removed when extra addresses are required for later changes. The only cost is the extra memory used and slower execution.

When re-assembly is undesirable or impossible a PATCH is recommended. This involves a jump to some external address, where the extra instructions are placed, followed by a "jump back" to the address just below the first "jump out". The cost is usually 6 bytes (2 jumps) plus the inserted code. In the program below a CLR COUNT instruction is needed just after STX MEMADD. Modify the program below to patch in the extra instruction assuming that COUNT is address 00FF and that addresses 0680 - 068F are available. Write both the assembly language instructions and the machine code for the patch.

```
0600 CE 134E         LDX    #$134E
0603 FF 0620         STX    MEMADD
0606 FE 0620         LDX    MEMADD
0609 08              INX

     0620    MEMADD EQU    $0620
```

```
0600 CE 134E         LDX    #$134E
0603 7E 0680         JMP    PATCH
0606 FE 0620         LDX    MEMADD
0609 08              INX
0680                 ORG    $0680
0680 FF 0620 PATCH   STX    MEMADD
0683 7F 00FF         CLR    COUNT
0686 7E 0606         JMP    $0606
```

The problem below presents a condition where memory locations for a patch are very limited. Assume that 5 bytes are available (0470 - 0474). The instruction CLR B is now needed between the first 2 instructions. In your solution show assembly language and machine code for changes made. If you are stuck, look at the hint in the first line of the answer.

```
0400 BD 1F00          JSR     TERMIN
0403 84 5F            AND A   #$5F
0405 81 4C            CMP A   #'L
```

Hint. Use branch rather than jump instructions.

Calculations

(1) 0470 - 0405

```
  FFFF
 -0405
  FBFA
 +   1
  FBFB
 +0470
  006B
```

```
0400 BD 1F00          JSR     TERMIN
0403 20 6B            BRA     PATCH
0405 81 4C    BACK    CMP A   #'L
0470                  ORG     $0470
0470 5F       PATCH   CLR B
0471 84 5F            AND A   #$5F
0473 20 90            BRA     BACK
```

(2) 0405 - 0475

```
  FFFF
 -0475
  FB8A
 +   1
  FB8B
 +0405
  FF90 ----> 90
```

Since only 5 locations are available branch instructions (2 bytes per branch) would just fit. Such situations are quite common when modifying old programs, particularly if source listings are unavailable.

The previous example shows how the program counter contents, when added to the branch offset, produces the address of the next instruction to be executed, e.g.,

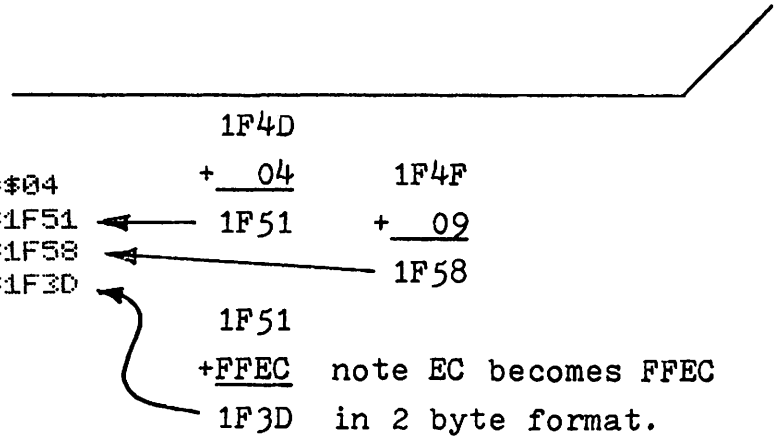
```
0405 = PC
+ 6B = branch offset
0470 = new address (where PATCH begins)
```

Reverse branching calculation is slightly different. Since 90 is a negative value, its 2 byte equivalent is then FF90

```
0475 = PC
+FF90 = branch offset (2 byte format)
0405 = new address (BACK)
```

Given the following machine code, convert it to assembly language producing absolute rather than symbolic addresses. Appendix D gives the instruction for each operation code.

```
1F49 81 04
1F4B 27 04
1F4D 8D 09
1F4F 20 EC
```




```
1F49 81 04          CMP A  #$04
1F4B 27 04          BEQ    $1F51
1F4D 8D 09          BSR    $1F58
1F4F 20 EC          BRA    $1F3D
```

Only negative values must be preceded by FF in 2 byte format.

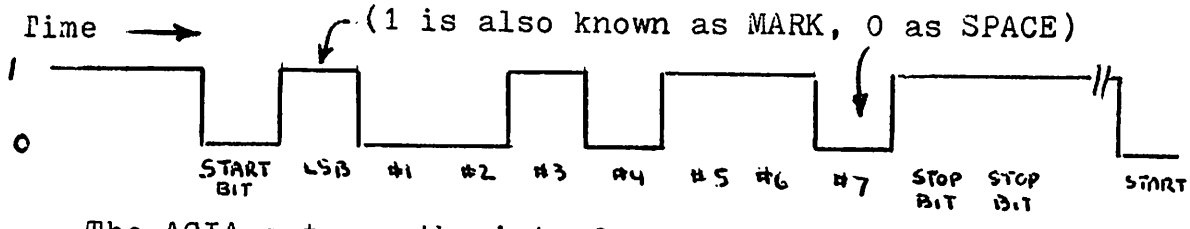
note EC becomes FFEC in 2 byte format.

If more practice is needed, there are lots of listings in the last half of this workbook.

- ACIA -
ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER

A computer, to perform any useful function, must be able to communicate with the "outside world", that is to and from external devices such as keyboards, printers, teletypes, remote computers, etc. Two forms of information transfer are available, serial and parallel. Parallel format, in which 8 bits are transferred at one time, requires 8 external data lines, plus control lines. For transmission of data beyond several hundred feet the large number of wires in a cable makes this parallel transmission impractical. In such cases serial transmission is preferable. For data transmission over a telephone line serial format is essential, since only one channel is available.

In serial format data is transmitted at a predetermined data rate, one bit after another. Each character or byte (usually 8 bits) is self contained, preceded by a start bit (always 0) and terminated by one or two stop bits (always 1). In between successive characters the signal remains in the 1 state, if there is a pause. A typical character is seen below.

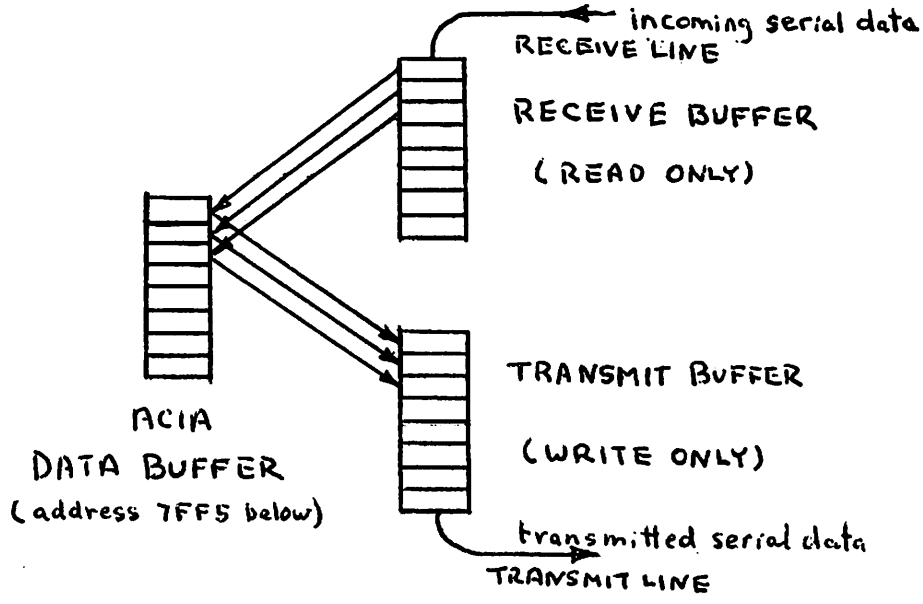


The ACIA acts as the interface between the serial device and the computer, communicating with the serial device in serial format and with the computer in parallel format.

Associated with the ACIA are 2 consecutive memory addresses, the lower one (even) controlling and indicating the status of the ACIA and the higher one (odd address) containing data transmitted or received by the ACIA. The actual addresses are usually in the top half of memory and are assigned by the hardware designer.

Contd.

Let's look at the Data Buffer first, assuming an address of 7FF5 for the ACIA Data Buffer "SERBUF". This single buffer services 2 internal buffers, receiving data from the "read only" RECEIVE BUFFER, and transmitting data to the "write only" TRANSMIT BUFFER. The same address is used for both buffers (see below). Hence the instruction LDA A SERBUF automatically gets its data from the RECEIVE BUFFER, while STA A SERBUF automatically passes its data to the TRANSMIT BUFFER.



Write an instruction which sends data, now in ACC A to the ACIA where it will be automatically put into serial form and transmitted to some external device.

STA A $7FF5 All that for one instruction!

Symbolic addresses are preferable when working with the ACIA. The statement

```
SERBUF EQU $7FF5
```

directs the assembler to substitute 7FF5 for the symbolic address SERBUF. To improve readability of programs it is usual practice to place all "EQU" assembler directives at the beginning of a program.

Address 7FF4 is known as the Control and Status Register, described in detail later in this chapter. Arbitrarily it is called SERCSR (SERial Control and Status Register).

7-2

Write an instruction to read serial data from the ACIA into ACC B. Assume previous symbolic definition of the Data Buffer.

LDA B SERBUF

Note that if STA A SERBUF
LDA A SERBUF
is executed, the data in ACC A will normally change since data is stored in the TRANSMIT buffer but loaded from the RECEIVE buffer, even though both carry the same symbolic address SERBUF.

7-3

If serial data is being received by the ACIA, some method is necessary to inform the computer when parallel data is ready. If data is read too soon it would be erroneous; if too late it could be lost, since the ACIA has only one 8 bit RECEIVE buffer where parallel data is stored after being formed from the incoming serial bit stream. At high serial data rates, e.g. 9600 bits/sec, the "lifetime" of data in the RECEIVE buffer is approximately 1 millisecond, after which it is overwritten by the next byte.

When an incoming data byte is ready, bit #0 of the Status Register (7FF4) automatically changes from 0 to 1. The AND or BIT instructions permit us to examine this bit #0, or "READY" bit, of the ACIA Receiver. It is normal practice to to test this bit in a looping manner, exit from the loop taking place when bit #0 = 1, that is when data is ready.

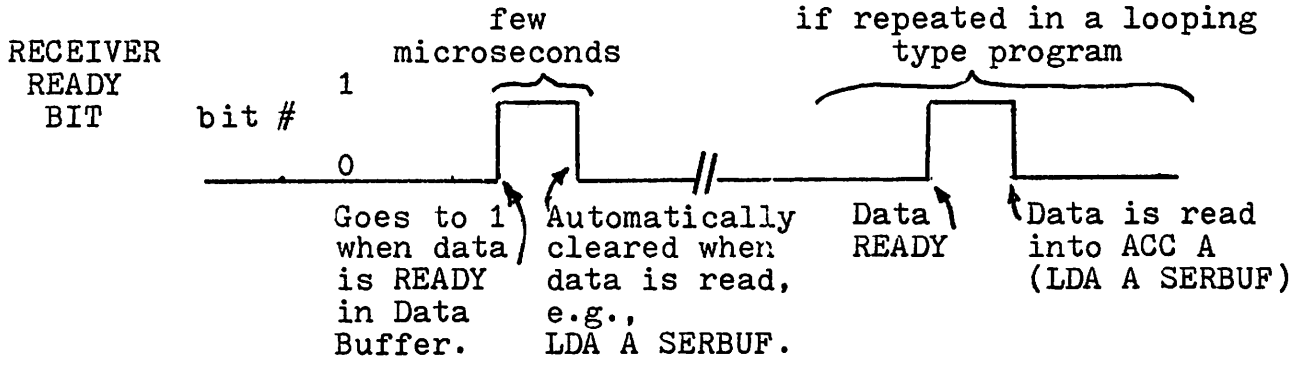
Write the instructions to examine bit #0 of the Status Register. (No branching yet.)

```
LDA A  SERCSR
AND A  #$01
```

Now add instructions to cause continuous testing of bit #0 until data is ready, whereupon the data is to be transferred to ACC A.

```
INLOOP LDA A  SERCSR
       AND A  #$01
       BEQ    INLOOP
       LDA A  SERBUF
```

Reading of the data from the RECEIVE buffer, SERBUF, clears the READY bit, sometimes referred to as a READY FLAG or DONE FLAG. A timing diagram of these events is shown here.



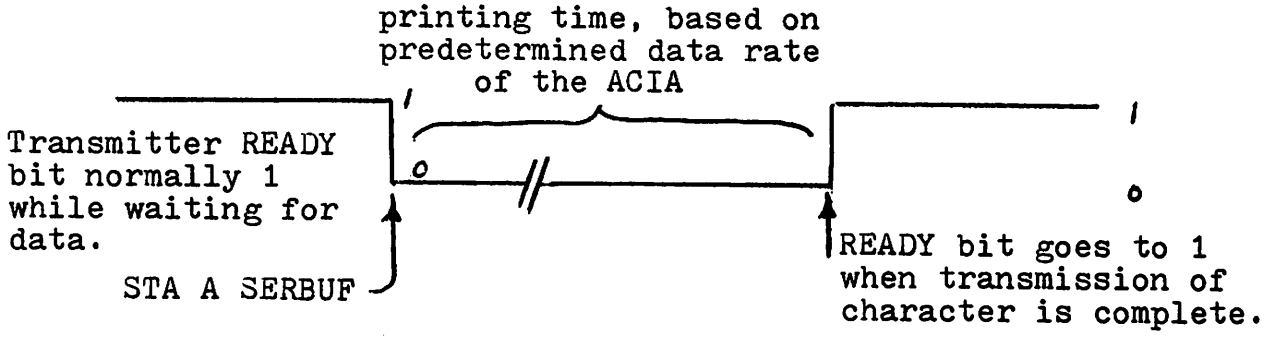
Although the rate of transmitting and receiving data bits is fixed there may be long time gaps between successive characters. Hence the term "asynchronous" in the ACIA, meaning no specified number of characters per second.

Data to be transmitted in serial form by the ACIA should not be transferred to the ACIA's TRANSMIT data buffer until this buffer is empty and therefore ready to accept a new byte. Bit #1 of the Status Register is the transmitter's READY bit. When in the 1 state, it denotes this READY condition.

Write a short program to put the byte now in ACC A into the TRANSMIT buffer when the transmitter is READY. Warning: Don't destroy data now in ACC A while testing for the READY condition.

```
OLOOP   LDA B   SERCSR
        AND B   #$02        TX READY?
        BEQ     OLOOP
        STA A   SERBUF      OUT TO TX
        END
```

The use of ACC B preserves the data in ACC A



Note that the transmitter, while dormant, is normally READY, waiting for data from the computer. In contrast, the receiver in the dormant state is normally not READY, since it is waiting for new serial data from the external device.

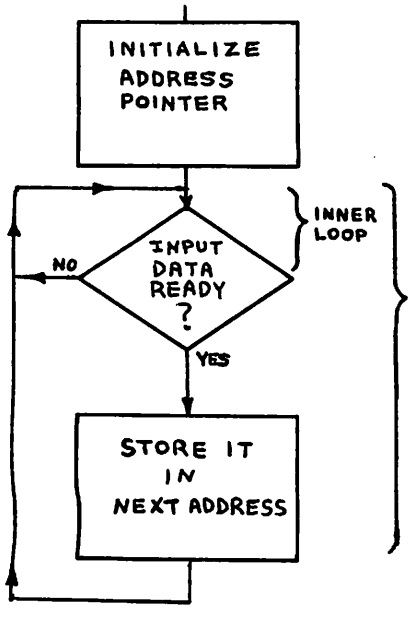
Now write a series of instructions to echo serial data from the ACIA RECEIVE line out on the ACIA's TRANSMIT line.

```
0200                    ORG     $0200
                *
                *
                *
     7FF4       SERCSR  EQU     $7FF4
     7FF5       SERBUF  EQU     $7FF5
                *
0200 B6 7FF4    INLOOP  LDA A   SERCSR
0203 84 01              AND A   #$01        RECEIVER READY?
0205 27 F9              BEQ     INLOOP
0207 B6 7FF5            LDA A   SERBUF      GET CHAR IN A
020A F6 7FF4    OLOOP   LDA B   SERCSR
020D C4 02              AND B   #$02        TX READY?
020F 27 F9              BEQ     OLOOP
0211 B7 7FF5            STA A   SERBUF      OUT TO TX
                        END
```

This is often known as an ECHO routine, permitting data which is entered on the keyboard to be viewed by the user.

To make this program more readable, the instruction AND A #$01 could be replaced by AND A #RXREDY, if RXREDY EQU $01 is included in the above definitions. Similarly AND B #$02 could be replaced by AND B #TXREDY.

Sometimes data, received by the ACIA must be stored, byte by byte, in memory. Flow chart and write a program to do this, the first byte going into address 1000. For now assume no end to this looping type program.




```
            LDX     #$1000-1
            STX     MEMADD      SET UP ADDRESS POINTER
MORTES      LDA A   SERCSR
            AND A   #$01        RX DATA READY?
            BEQ     MORTES
            LDA A   SERBUF      GET CHAR
            LDX     MEMADD
            INX
            STX     MEMADD      AND NEW ADDRESS
            STA A   X           AND STORE CHAR
            BRA     MORTES      BACK FOR MORE
MEMADD      RMB     2
            END
```

Here we see an inner loop testing the READY bit and an outer loop storing data. This is known as a "nested" loop format.

Modify your program such that receipt of 5A will cause storage of this byte, then exit from the loop. Show changes only.

Before		After		
BRA	MORTES	CMP A	#$5A	IS IT Z?
		BNE	MORTES	

If your modification looked like this:

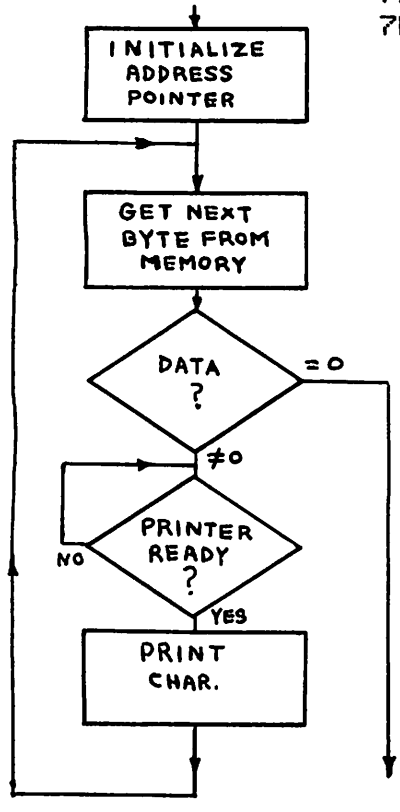
```
                    CMP A   #$5A      IS IT Z?
                    BEQ     NEXT
                    BRA     MORTES
                NEXT
```

note that a conditional branch (BEQ NEXT) followed by an unconditional branch (BRA) can usually be replaced by a single branch instruction (BNE MORTES) of the opposite sense (BNE vs BEQ).

Although the ASCII code for Z is 5A some terminals produce "mark parity", that is the leading bit is always set, resulting in DA rather than 5A. Other terminals may produce "space parity" (leading bit is zero) or odd or even parity, discussed a few pages later.

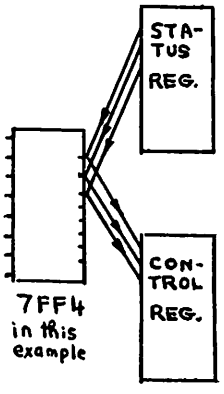
The computer when connected via the ACIA to some output device such as a printer or CRT terminal could send a specific message to the computer operator.

Flow chart and write a program to output the message BAD HEX CHAR to such an output device via the ACIA. Terminate the message with a null.




```
7FF4    SERCSR EQU    $7FF4
7FF5    SERBUF EQU    $7FF5
        *
               LDX    #MESBAD-1
               STX    MEMPNT    INIT MESS POINT
PRTMOR         LDX    MEMPNT
               INX
               STX    MEMPNT    GET POINT ADDRESS
               LDA A  X         GET CHAR FROM MEM
               BEQ    ALLDUN    QUIT IF NULL
OUTEST         LDA B  SERCSR
               AND B  #$02      OUTPUT DEVICE READY?
               BEQ    OUTEST    NOT YET
               STA A  SERBUF    YES OUTPUT IT
               BRA    PRTMOR
MEMPNT         RMB    2
MESBAD         FCC    /BAD HEX CHAR/
               FCB    0
ALLDUN
```

To operate the ACIA correctly the data rate at the receiving end must be within 1 or 2% (5% would produce errors) of the transmitted data rate. Hence the frequency of external oscillator which determines the basic data rate for each ACIA is usually crystal-controlled, as in modern electronic watches.



Selection of data rates and control operations are possible via the Control Register, a "WRITE ONLY" register which shares the same address as the "READ ONLY" Status Register. The diagram at the left depicts these registers, assuming 7FF4 as the assigned address. Hence LDA A $7FF4 reads from the Status Register, while STA A $7FF4 stores in the Control Register. The common symbolic address in previous examples has been SERCSR.

The data rate of the ACIA is determined by dividing the external oscillator's frequency by 64, 16 or 1, under control of bits #0 and 1 of the Control Register (see App. E1). For example, if bit #1 is 0 and bit #0 is 1 (÷16 mode) an oscillator frequency of 9600 bps would produce a data rate of 9600/16 = 600 bps.

Assuming that all other control bits are correctly set ensure that the ACIA will operate at a data rate of 300 bps when the oscillator frequency is 19200 Hz (cycles/sec). Since the Control Register cannot be read to be modified, assume that it is updated from ACIACR, a symbolic address in memory.

```
0100 B6 738E        LDA A  ACIACR   GET ORIGINAL STATUS
0103 84 FE          AND A  #%11111110  CLEAR BIT 0
0105 8A 02          ORA A  #%00000010  SET BIT 1
0107 B7 738E        STA A  ACIACR   UPDATE ORIGINAL
010A B7 7FF4        STA A  SERCSR
```

19200/300 = 64

Therefore bit #1 = 1) See
bit #0 = 0)- Appendix
in the Control Register) E.

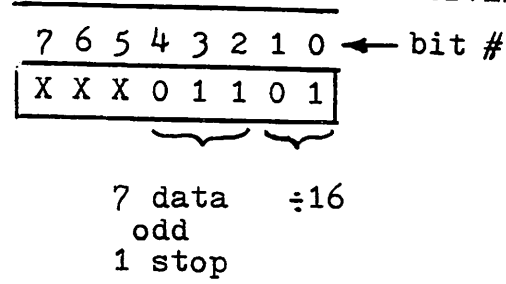
If both bits are 1 RESET takes place. This is necessary when power is first turned on, before changing speed, parity, etc.

Bits 2, 3 and 4 (see Appendix E) determine the number of data bits and stop bits of the data format. It also determines the parity options for the data. Parity control determines whether each transmitted data byte carries an even, odd or unspecified number of ones, bit #7 of the data being modified to produce odd or even parity.

The number of data bits and stop bits, plus parity options must be agreed upon for both ends of the data link. Although programmable, they are not usually changed once a data link is set up.

Without disturbing unspecified Control Register bits, set the ACIA for 1200 bps operation using a 19200 bps oscillator. The data formed is to be 7 data bits plus 1 odd parity bit plus 1 stop bit. Again use ACIACR as the original for the Control Register.

```
0100 B6 738E        LDA A  ACIACR   GET ORIGINAL STATUS
0103 84 ED          AND A  #%11101101  CLEAR BITS 1 AND 4
0105 8A 0D          ORA A  #%00001101  SET BITS 0,2 AND 3
0107 B7 738E        STA A  ACIACR   UPDATE ORIGINAL
010A B7 7FF4        STA A  SERCSR   CHANGE CONTROL REGISTER
```



7 data ÷16
odd
1 stop

For your first few programs, which are not part of a larger program, simply place the desired value in the Control Register e.g.

```
LDA A #%00001101
STA A SERCSR
```

Serial data processed by the ACIA essentially follows the RS-232-C Specifications of the Electronic Industries Association (EIA). Voltage levels, source and load resistances, connector type and pin assignments for data and control signals are contained within this specification. Some of these control signals are produced by the ACIA for the serial device. Others are produced by the serial device for the ACIA.

One control signal is RTS (Request To Send), which is produced by the ACIA when requesting permission of the serial device, a printer perhaps, to send data to it. This signal is active when low hence is called $\overline{RTS}$, the bar over RTS indicating inversion, that is when $\overline{RTS}$ = 1, RTS = 0. $\overline{RTS}$ is determined by Control Register bits #6 and 5.

The usual response by a serial device (printer) upon receiving $\overline{RTS}$ = 0 is to activate a control line to the ACIA called $\overline{CTS}$ (Clear To Send), also active when low.

This exchange of control signals, usually preceding data transmission, is often called "hand shaking" and can be used to permit data transfer only when a device is turned on and operational. The $\overline{RTS}$ line can alternately be used as a control line without feedback ($\overline{CTS}$ is ignored), perhaps controlling a function in an external device.

Control Register bits #7, 6 and 5 remain to be discussed. Bit #7 controls receiver "Interrupt" operations (Chapter 11) and is assumed to be 0 for now. Similarly bit #5 is assumed to be 0 since it controls transmitter "Interrupt" and "Break" operations. With bit #5 = 0, bit #6 controls the $\overline{RTS}$ line; $\overline{RTS}$ = 0 when bit #6 = 0, and 1 when bit #6 = 1. See Appendix E for details.

The following program is to:

(a) initialize the ACIA for operation with:
- 7 data bits, even parity and 1 stop bit.
- data rate of 600 bps when the oscillator frequency is 38400 bps.

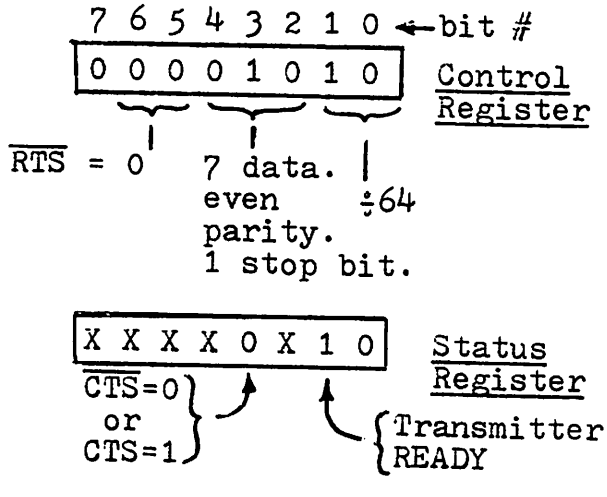
(b) set $\overline{RTS}$ = 0.

(c) send the ASCII code ACK (acknowledge) after the external device (printer) clears $\overline{CTS}$.

Contd.

```
      7FF4     SERCSR EQU    $7FF4
      7FF5     SERBUF EQU    $7FF5
      738E     ACIACR EQU    $738E
               *
0100 B6 738E          LDA A  ACIACR
0103 84 8A            AND A  #%10001010
0105 8A 0A            ORA A  #%00001010
0107 B7 738E          STA A  ACIACR   UPDATE ORIGINAL
010A B7 7FF4          STA A  SERCSR
010D F6 7FF4 NOTYET   LDA B  SERCSR
0110 C4 0A            AND B  #%00001010
0112 C1 02            CMP B  #%00000010
0114 26 F7            BNE    NOTYET
0116 86 06            LDA A  #$06
0118 B7 7FF5          STA A  SERBUF
```

Explain the function of the 4 instructions starting with LDA B SERCSR



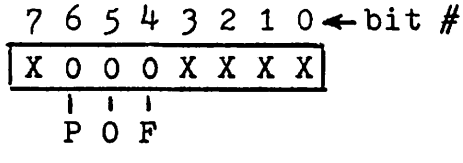
LDA B SERCSR and
AND B #%00001010 "expose"
Status Register bits #3 & 1.
CMP B #%00000010 tests for
0 in bit #3 ($\overline{CTS}$=0) and
1 in bit #1 (Tx READY).
BNE NOTYET branches back if
either condition is not met.

Returning to the Status Register, other bits not yet discussed are:

- Bit #2 - $\overline{\text{Data Carrier Detect}}$ or $\overline{\text{DCD}}$ an input to the ACIA from a "modem" used to transmit serial data over a telephone line. $\overline{\text{DCD}}$ = 1 if loss of tone occurs on the telephone line.
- Bit #4 - Framing Error goes to 1 when a stop bit is missing, usually due to an erroneous start bit.
- Bit #5 - Receiver Overrun - goes to 1 when data is lost due to too slow reading of the Data Buffer. It is cleared by reading the Data Buffer.
- Bit #6 - Parity error, goes to 1 when the parity of the received data differs from that expected, based on the Control Register contents.
- Bit #7 - Interrupt Request state (Chapter 11).

Write a few instructions to ensure that the Framing Error, Receiver Overrun and Parity Error bits are all normal (zero). If one or more is wrong, branch to ERROR.

```
7FF4     SERCSR EQU    $7FF4
7FF5     SERBUF EQU    $7FF5
         *
B6 7FF4         LDA A  SERCSR
84 70           AND A  #%01110000  CHECK FOR 3 TYPES OF ERROR
26 59           BNE    ERROR
```

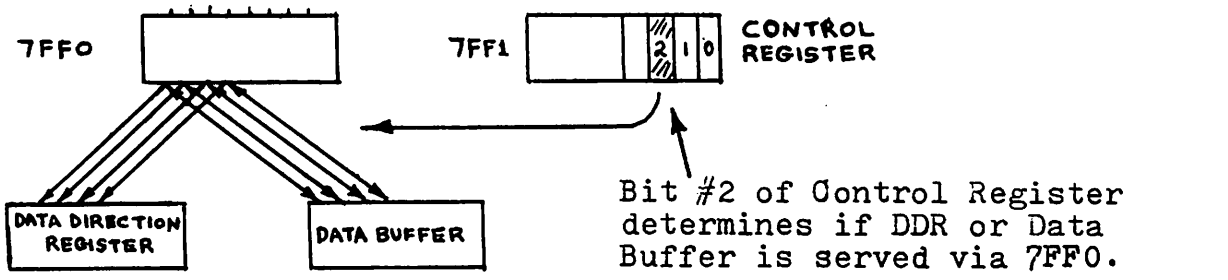


- PIA -
PERIPHERAL INTERFACE ADAPTER

In the previous chapter we worked with the ACIA which transmits and receives serial data in a fixed format at a predetermined rate. This chapter involves the Peripheral Interface Adapter (PIA), a device which transmits and receives data in parallel form at an unspecified data rate.

The PIA is comprised of 2 almost identical sections, A and B, each capable of transmitting or receiving 8 bits of data. A block diagram of the "A" half of the PIA is shown below. For each section there is a Control Register (CR) and a Data Buffer, both having similar functions to those in the ACIA, plus a Data Direction Register (DDR) which determines which bits of the Data Buffer are inputs and which are outputs. Both the Data Buffer and the Data Direction Register share the same official memory address, the selection between the two depending on the state of bit #2 of the Control Register.

Assume address 7FF0 for the DDR and Data Buffer for the A half of the PIA. Automatically its Control Register address would be 7FF1. For the "B" half of the PIA the addresses would be 7FF2 and 7FF3 (Data Buffer and DDR = 7FF2, CR = 7FF3).



Let's assign symbolic addresses to these two memory addresses, PIABFA being the "A" half Data Buffer (and DDR too) at address 7FF0. Similarly PIACRA would be the "A" half of Control Register at 7FF1. For the "B" half the corresponding symbolic addresses would be PIABFB (Data Buffer and DDR) at 7FF2, and PIACRB (Control Register) at 7FF3.

Contd...

As noted in the previous diagram, if bit #2 of PIACRA = 0, then data destined for PIABFA goes to the "A" Data Direction Register. If this bit #2 = 1, the data will go to the "A" Data Buffer.

The Data Direction Register stores 8 bits, each bit independently controlling the data direction for the corresponding bit of the Data Buffer; 1 = output, 0 = input.

Write the instructions to ensure that all PIA data lines for the "A" half of the PIA will be input lines. Note that the first task is to address the Data Direction Register, via bit #2 of the Control Register.

```
                *
     7FF0       PIABFA EQU    $7FF0
     7FF1       PIACRA EQU    $7FF1
                *
0100 B6 7FF1            LDA A  PIACRA
0103 84 FB              AND A  #%11111011  CLEAR BIT 2 TO ACCESS DDR
0105 B7 7FF1            STA A  PIACRA
0108 7F 7FF0            CLR    PIABFA   SET A HALF FOR INPUT
```

The routine in the previous frame would normally be found within a RESET program which is automatically executed when the microprocessor power is first applied or when the RESET button is depressed. More details on such initializing operations are contained in the Interrupt Chapter.

Write the instructions for a RESET routine to set up the "A" half of the PIA for input and the "B" half for output. This routine should leave the PIA ready to load and store data.

```
                *
     7FF0       PIABFA EQU    $7FF0
     7FF1       PIACRA EQU    $7FF1
     7FF2       PIABFB EQU    $7FF2
     7FF3       PIACRB EQU    $7FF3
                *
0100 B6 7FF1 AHALF  LDA A  PIACRA
0103 84 FB          AND A  #%11111011  CLEAR BIT 2 TO ACCESS DDR
0105 B7 7FF1        STA A  PIACRA
0108 7F 7FF0        CLR    PIABFA   SET A HALF FOR INPUT
010B 8A 04          ORA A  #%00000100  BIT 2 = 1 FOR DATA
010D B7 7FF1        STA A  PIACRA
0110 F6 7FF3 BHALF  LDA A  PIACRB
0113 84 FB          AND A  #%11111011  CLEAR BIT 2 TO ACCESS DDR
0115 B7 7FF3        STA A  PIACRB
0118 86 FF          LDA A  #%11111111
011A B7 7FF2        STA A  PIABFB   SET B DDR FOR OUTPUT
011D B6 7FF3        LDA A  PIACRB   GET CR AGAIN
0120 8A 04          ORA A  #%00000100  BIT 2 = 1 FOR DATA
0122 B7 7FF3        STA A  PIACRB
```

8-3

Assuming that the B half of the PIA is already initialized for output (see previous frame), set bit #5 and clear bit #3 of Data Buffer B, without disturbing other Data Buffer bits. From now on assume PIA Register definition (PIABFA EQU $7FF0 etc.), unless otherwise requested.

```
LDA A  PIABFB
ORA A  #%00100000   SET BIT 5
AND A  #%11110111   CLEAR BIT 3
STA A  PIABFB
```

The PIA could be controlling a machine tool, with the changes in bits #3 and #5 representing control signals for the next machine process.

8-4

What is the state of bit #2 of PIACRB during the previous frame?

Bit #2 of PIACRB = 1 permitting communication with the Data Buffer rather than the Data Direction Register.

8-5

The PIA could be used with a 6800 microcomputer in an automobile sensor and alarm system. Assume INDATA as Data Buffer A, at address 7FF0. Also assume the following bit assignments for INDATA.

Contd...

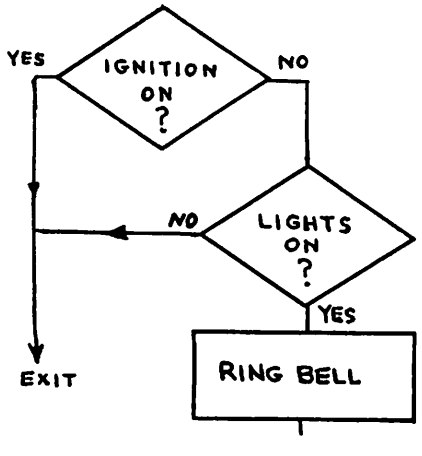
The input Buffer, INDATA, has the following bit assignments.

Bit #	Function	Status if 0	Status if 1
0	Seat Belt Monitor	disconnected	fastened
1	Door Monitor	closed	opened
2	Oil Pressure Monitor	low	normal
3	Ignition Monitor	ignition off	ignition on
4	Gear Shift Monitor	park/neutral	all others
5	Engine Monitor	not running	running
6	Day/Night Monitor	night	day
7	Headlight Monitor	lights off	lights on

The output Buffer, OUTDAT, has the following bit assignments.

Bit #	Function	Status if 0	Status if 1
0	Buzzer	off	on
1	Bell	off	on
2	Panel Alarm Light	off	on
3	Starter Control	starting disabled	starting enabled

Flow chart and write the instructions to ring the bell if the ignition is off and the headlights are on. (I wish that I had that on my car.) Assume previous initialization of the PIA for input on Buffer A and output on Buffer B.




```
                    *
0100 B6 7FF0 CARCHK LDA A   INDATA
0103 85 08          BIT A   #%00001000   IGN?
0105 26 49          BNE     NOBELL
0107 85 80          BIT A   #%10000000   LIGHT?
0109 27 45          BEQ     NOBELL
010B B6 7FF2        LDA A   OUTDAT
010E 8A 02          ORA A   #%00000010   BELL?
0110 B7 7FF2        STA A   OUTDAT    RING BELL
                    *
                    *
0150 .              NOBELL .
```

This time permit the car to be started if and only if:

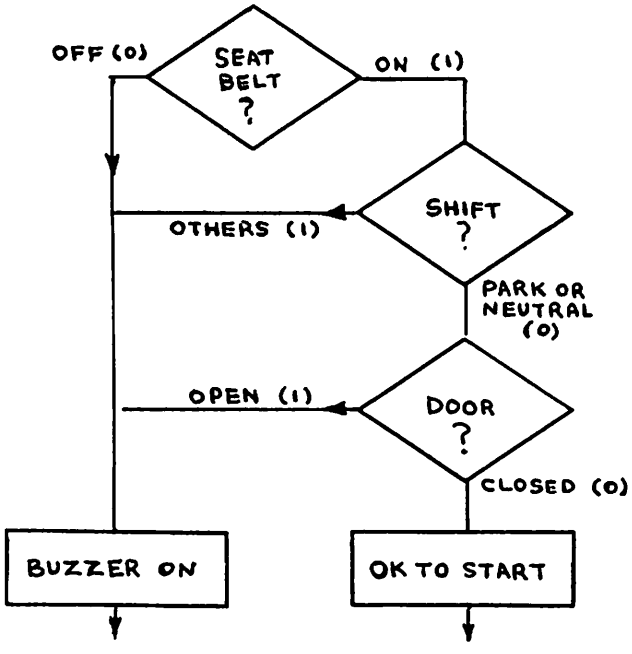
(a) seat belt is fastened and

(b) gear shift is in Park or Neutral and

(c) door is closed.

otherwise turn on the buzzer.

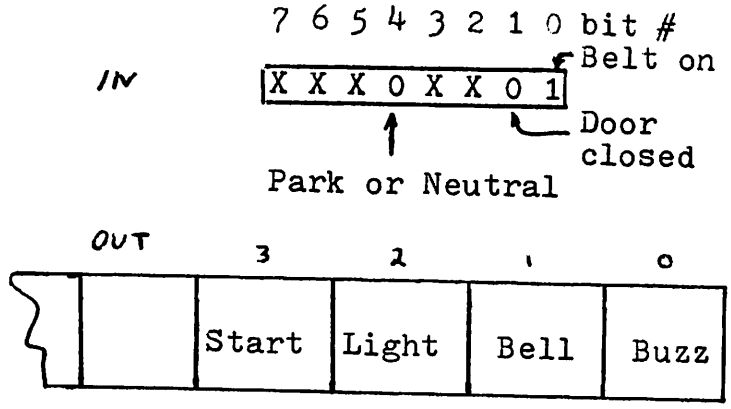
First flow chart your solution.



Your order of checking the functions may correctly be different. The order shown here leads to slightly easier testing as seen in answer in the next frame.

Now write the program, preferably using the flow chart shown in the previous frame.

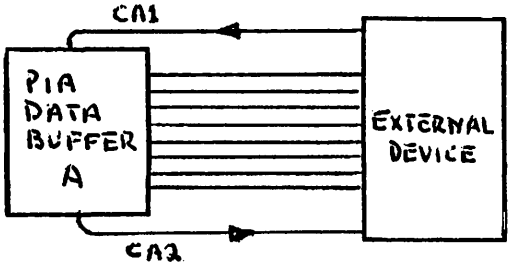
```
0122 B6 7FF0 TESCAR LDA A  INDATA
0125 85 01          BIT A  #%00000001  BELT ON?
0127 27 04          BEQ    BUZZ
0129 85 12          BIT A  #%00010010  GEAR SHIFT AND DOOR?
012B 27 0A          BEQ    OKTOGO
012D B6 7FF2 BUZZ   LDA A  OUTDAT
0130 8A 01          ORA A  #%00000001
0132 B7 7FF2        STA A  OUTDAT      BUZZ
0135 20 08          BRA    DONE
0137 B6 7FF2 OKTOGO LDA A  OUTDAT
013A 8A 08          ORA A  #%00001000  OK TO START
013C B7 7FF2        STA A  OUTDAT
013F 20 E1   DONE
```

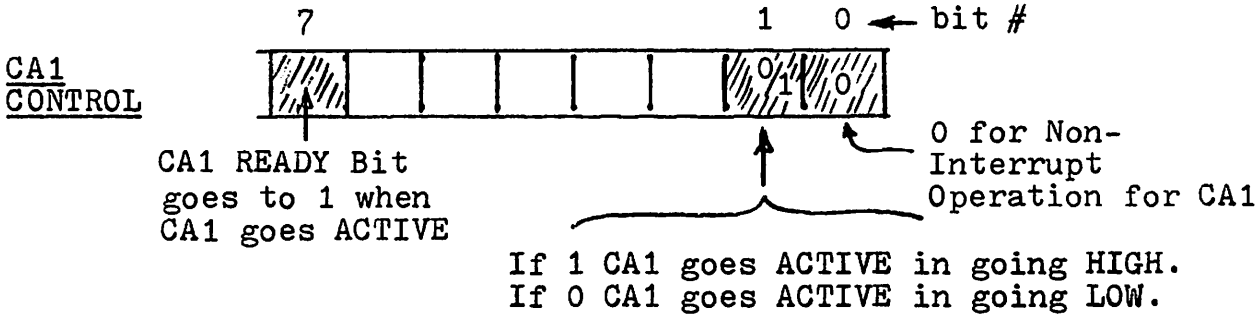


By grouping the Gear Shift and Door checks together the single instruction BIT A #$00010010 will cause a branch via BEQ OKTOGO if and only if both bits are 0.

Transfer of data between the PIA and an external device takes place at an unspecified rate; hence control lines are needed between the PIA and the external device to indicate to the PIA when the data is ready and to the external device when the data has been read. This provides a "hand shaking" linkage similar to that possible via RTS and CTS in the ACIA.

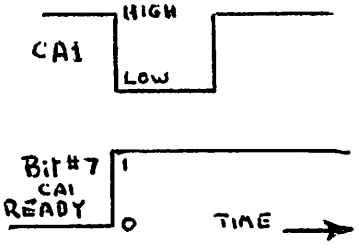
For the A half of the PIA two control lines, CA1 (input to the PIA) and CA2 (input or output) are available. CA1 could inform the PIA, acting as a data receiver, that data is now available. When this data is read by the PIA, CA2 could inform the external device that data has been read; therefore another byte could be placed on the data lines. CB1 and CB2 could perform similar functions for the B half. Both CA1 and CA2 are controlled by specific bits of Control Register A as shown below.



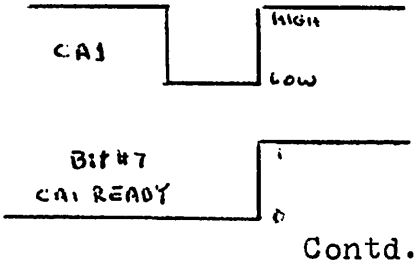


The 3 bits associated with CA1 are shown above. We are not using interrupt at this time; hence bit #0 = 0. Bit #1 determines whether CA1 sets the READY bit (#7) when CA1 goes LOW (if bit #1 = 0) or HIGH (if bit #1 = 1). The CA1 READY bit (also called IRQA1 in Motorola literature) indicates, when going to the 1 state, that CA1 has gone ACTIVE.

If bit #1 = 0 in the Control Register



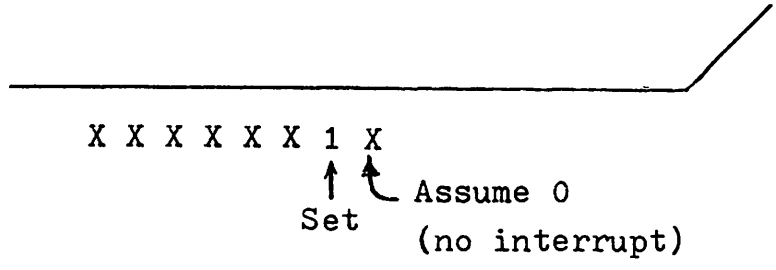
If bit #1 = 1 in the Control Register



Contd...

The PIA "READY" bit (similar to the ACIA "READY" bit) will be cleared automatically when data is read from the Data Buffer, e.g. LDA A PIABFA. Bit #7 of the Control Register is a READ ONLY bit, and therefore cannot be set or cleared by a STA A PIACRA instruction.

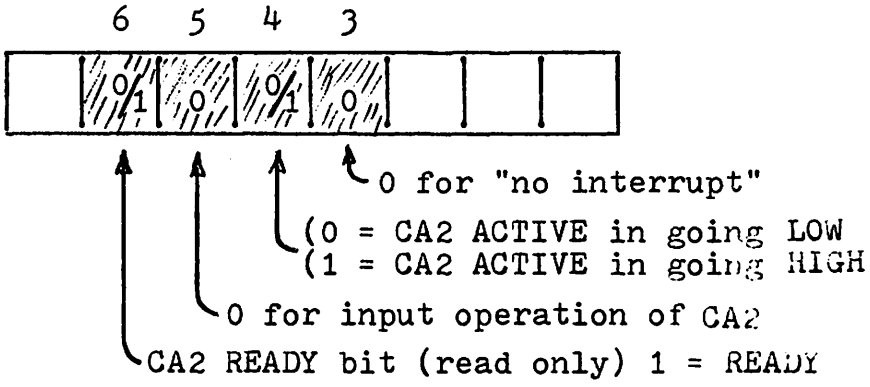
Initialize Control Register A so that CA1's READY bit is set when CA1 goes HIGH. Do not disturb the other Control Register bits.




```
0100 B6 7FF1        LDA A  PIACRA
0103 8A 02          ORA A  #%00000010  SET BIT 2
0105 B7 7FF1        STA A  PIACRA
```

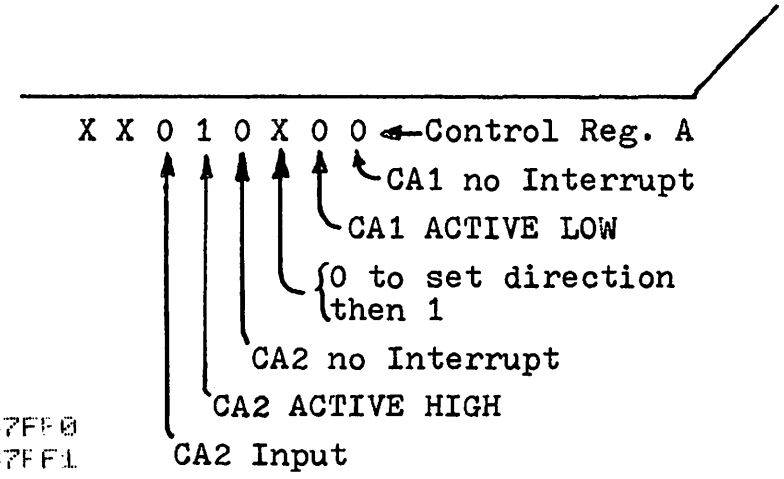
Note that it is the transition (LOW to HIGH or HIGH to LOW) which causes the input Control Lines to become ACTIVE, rather than the final level of these lines

When bit #5 of Control Register A = 0, CA2 also acts as an input line similar to CA1. Bit assignments for PIACRA are as follows.



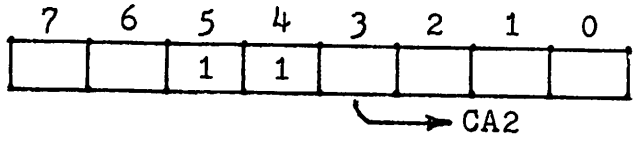
Bit #5 = 0 for input. Bits #4 and 3 behave the same as bits #1 and 0 for CA1.

Assume that both CA1 and CA2 are to be input control lines, CA1 being ACTIVE in going LOW and CA2 being ACTIVE in going HIGH. Write the instructions to produce this. Also set up the A Data Buffer for input operation.




```
     7FF0     PIABFA EQU    $7FF0
     7FF1     PIACRA EQU    $7FF1
     7FF2     PIABFB EQU    $7FF2
     7FF3     PIACRB EQU    $7FF3
0100 B6 7FF1          LDA A  PIACRA
0103 84 D0            AND A  #%11010000
0105 B7 7FF1          STA A  PIACRA   SET FOR  DDR
0108 7F 7FF0          CLR    PIABFA   INPUTS FOR A HALF
010B 8A 14            ORA A  #%00010100  DATA BUF NOW
010D B7 7FF1          STA A  PIACRA
```

There are 3 possible modes for CA2, acting as an output (bit #5 = 1). The first is seen when bit #4 = 1. CA2 will now act as an output line whose state will be determined by bit #3, (0 produces LOW, 1 produces HIGH).

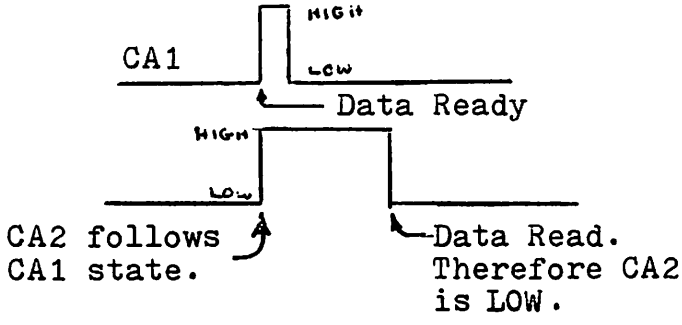


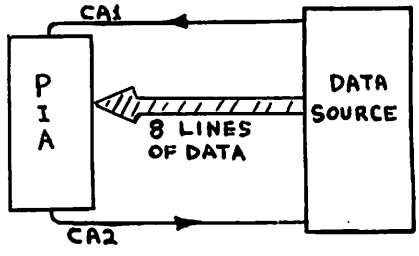
Assume that to communicate with some external device CA2 is to go to the HIGH state for 1 millisecond, then go LOW. Also assume that the instruction JSR MILSEC (subroutines will be covered in the next chapter) will cause a delay of 1.0 milliseconds. Write the necessary instructions assuming that CA2 is presently LOW.

```
      7FF0     PIABFA EQU    $7FF0
      7FF1     PIACRA EQU    $7FF1
      7FF2     PIABFB EQU    $7FF2
      7FF3     PIACRB EQU    $7FF3
0100 B6 7FF1          LDA A  PIACRA
0103 8A 38            ORA A  #%00111000  SET BITS 5, 4 AND 3.
0105 B7 7FF1          STA A  PIACRA    NOW CA2=1
0108 BD 0113          JSR    MILSEC    ONE MILLISEC DELAY
010B B6 7FF1          LDA A  PIACRA    MILSEC MAY USE ACC A
010E 84 F7            AND A  #%11110111  CLEAR BIT 3
0110 B7 7FF1          STA A  PIACRA    CA2=0
```

Such an output control signal on CA2 could be produced after data reception on the A half of the PIA to order the data source to change mode of operation. For lack of a better name let's call this the PROGRAMMED mode, since the state of CA2 is determined by program control.

CA2 may be used as an output control line in a "hand shaking" mode when bit #5 = 1 and bits 4 and 3 = 0. In this mode the A half acts as a data receiver. CA2 will go HIGH automatically when CA1 goes ACTIVE (HIGH in this example) and will go LOW automatically when Data Buffer A is read.

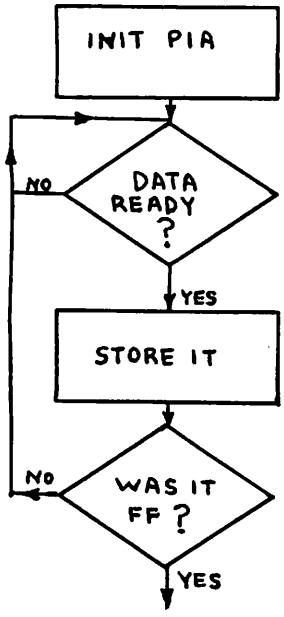




When CA2 goes LOW the external device will know that new data may be put on the data lines.

Flow chart and write the instructions to read the data from the external source via the PIA (A half) when CA1 goes HIGH, automatically indicating via CA2 that the data has been read. Store the data starting at 0800, terminating data storage after FF has been read and stored.

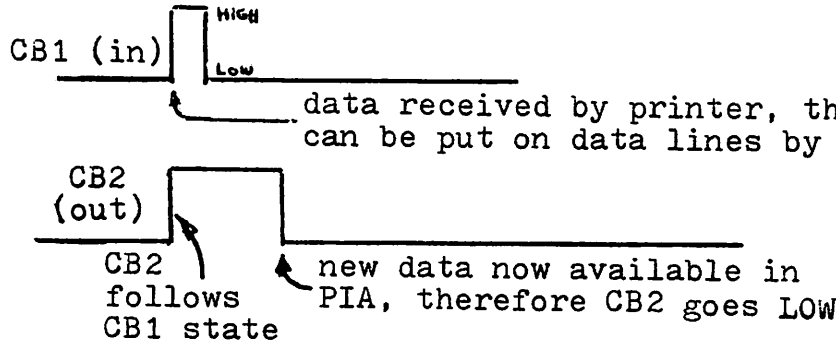
Contd...




```
*
PIAHAN LDX     #$0800-1
       STX     MEMPNT    INIT POINTER
       LDA A   PIACRA
       AND A   #%11100010  BITS 4,3,2,0 = 0
       STA A   PIACRA
       CLR     PIABFA    INPUT MODE NOW
       ORA A   #%00100110  SET BITS 5,2 AND 1
       STA A   PIACRA    DATA BUF NOW
INWAIT LDA A   PIACRA
       BPL     INWAIT    WAIT FOR READY FLAG
       LDA A   PIABFA    GET DATA
       LDX     MEMPNT
       INX
       STX     MEMPNT    GET STORE ADDRESS
       STA A   X         AND STORE DATA
       CMP A   #$FF
       BNE     INWAIT    NOT LAST DATA
HR     BRA     HR        ALL DONE SPIN FOREVER
MEMPNT RMB     2
```

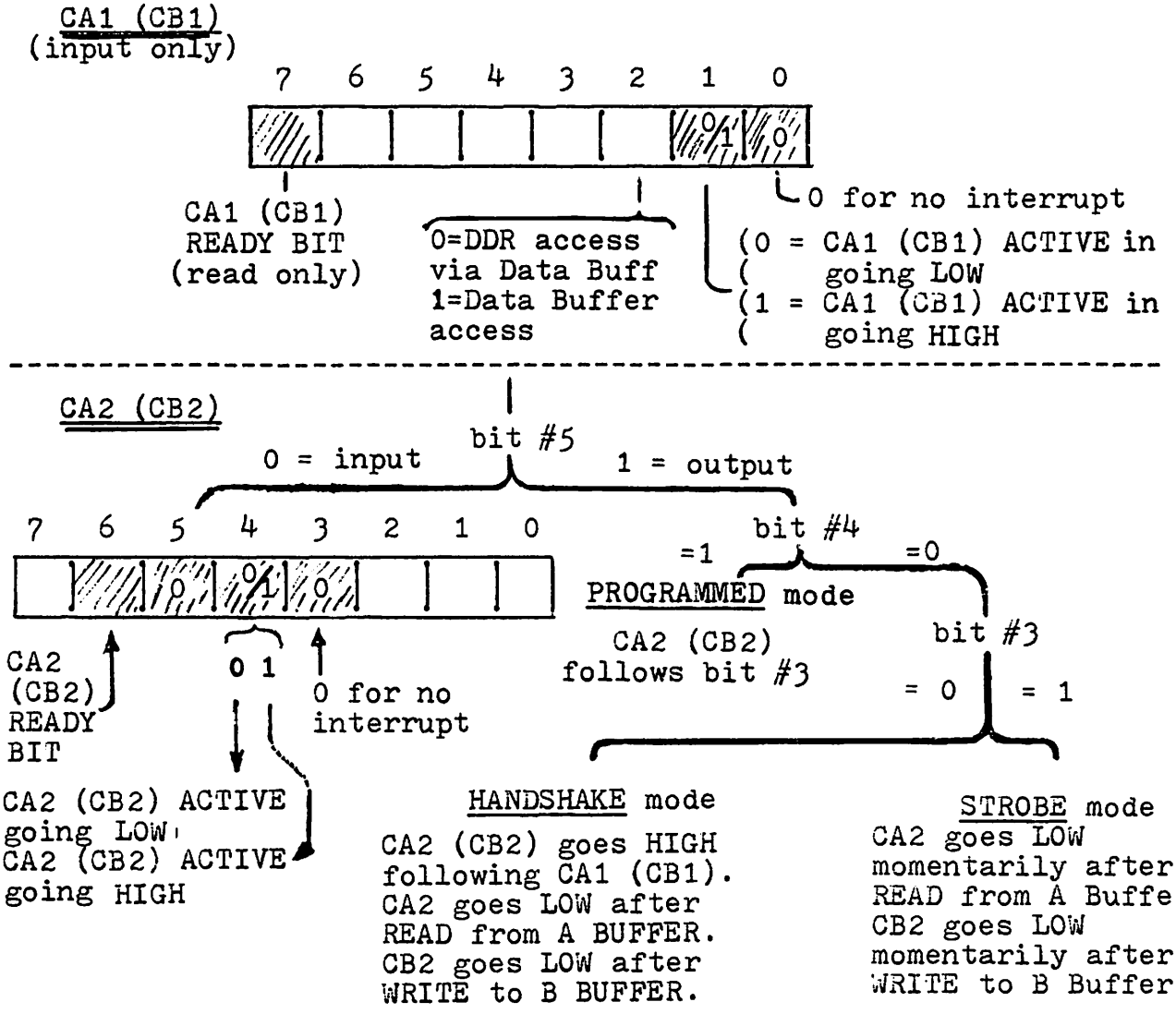

In the same hand shaking mode (bit #5 = 1, bits #4 and 3 = 0), the B half of the PIA acts as a transmitter. Here CB2 will go HIGH when CB1 goes ACTIVE (HIGH in this example) and will go LOW when data is written out (stored) in Data Buffer B.

Sketch timing diagrams for CB1 and CB2 indicating the reason or significance of each change. When working this out think of what information the PIA (transmitter) and the external device (e.g., printer) need to know to transmit data without loss of data or loss of time.

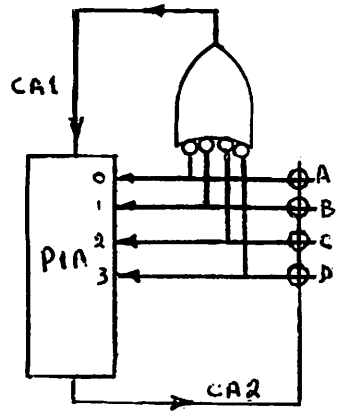


Again the hand shaking operation permits optimum data flow. Although the printer would not normally store more than 132 characters for one complete line of text, the data rate within this line could be as high as 50 000 characters/second, limited by the computer's clock and the number of instructions per loop.

One last mode, the STROBE mode is available when bit #5 = 1, bit #4 = 0, and bit #3 = 1. It is similar to the previous HANDSHAKE mode in that CA2 goes low when data is read (LDA A PIABFA) into the A Data Buffer. It differs in that CA2 automatically returns to the 1 state several microseconds (one instruction) later. Similarly, in the B half of the PIA, CB2 goes low when a write operation (STA A PIABFB) takes place and returns to the 1 state automatically, several microseconds later. This mode of operation releases CA1 and CB1 for other tasks, but assumes that data is always ready for the "A" half and that the external device is always ready to receive data from the "B" half. A summary of control line operations is shown below.



No answer is required in this frame.



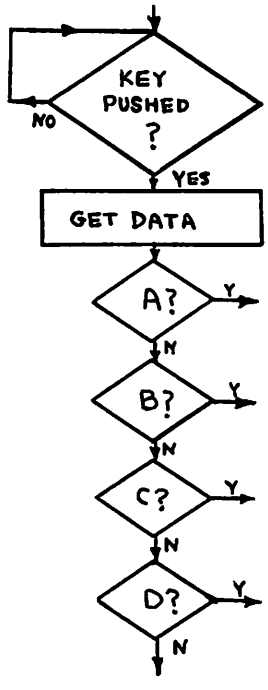
Here is an application of the PIA to detect which of the 4 keys, A, B, C or D was depressed. CA2 provides logic 0 to all 4 intersections, the depressed key passing on this 0 state to the appropriate input. The symbol at the top of the diagram is an "inverted input OR gate" whose output goes to the 1 state if one or more of the inputs go to 0. PIA lines 4 to 7 are not needed.

Write the initialization instructions for the PIA to set up CA1 as an input (ACTIVE high) and CA2 as an output, following bit #3. The Data Buffer should be set up as an input.

X X 1 1 0 % 1 1 0
CA2 output = bit #3
CA1 input active high

```
                * PIA PROG FOR FOUR KEY KEYBOARD.
                * CA2 IS OUTPUT TO SWITCHES. CA1 IS
                * INPUT TO PIA. DATA GOES TO LOW 4 BITS.
                *
0100 B6 7FF1 KEYPIA LDA A  PIACRA
0103 84 F2          AND A  #%11110010
0105 B7 7FF1        STA A  PIACRA   ACCESS DDR
0108 7F 7FF0        CLR    PIABFA   DATA INPUT MODE
010B 8A 36          ORA A  #%00110110
010D B7 7FF1        STA A  PIACRA   DATA MODE NOW
```

Now flow chart and write the instructions to branch to KEYA, KEYB, KEYC, or KEYD, corresponding to a depression of keys A, B, C or D.




```
TRYAGN LDA A   PIACRA
       BPL     TRYAGN    CA1 NOT UP YET
       LDA A   PIABFA    UP NOW
       AND A   #$0F      LOWER 4 BITS ONLY
       BIT A   #$01      KEY A HIT?
       BEQ     KEYA      YES.
       BIT A   #$02      KEY B HIT?
       BEQ     KEYB
       BIT A   #$04      KEY C HIT?
       BEQ     KEYC
       BIT A   #$08      KEY D HIT?
       BEQ     KEYD
```

For short tests this "brute force" method is acceptable. For longer checks, data table lookups should be used.

Whenever mechanical devices such as switches are used there exists a problem of contact bounce; that is the contacts may close, open, then close, several times within a few milliseconds of the first contact before settling down to a "closed" or ON condition. Data or signals from such a switch are highly unpredictable during this transient period, hence a timing loop of perhaps ten milliseconds should be introduced after the first contact detection, via CA1 or CA2 before the PIA Data Buffer is read.

Assuming a 1MHz (10^6 cycles/sec.) clock in the 6800 microprocessor, the number of microseconds per instruction executed can be determined from Appendix C under the ~ column denoting the number of machine cycles per instruction.

```
LDX #$0400
```

an immediate mode instruction, requires 3 cycles or 3 microseconds.

What is the execution time per loop in:

```
MORDEX DEX
       BNE    MORDEX
```

```
MORDEX DEX              4 CYCLES
       BNE    MORDEX    4 CYCLES
*
*                       8 CYCLES TOTAL
```

To get 10 msec., then the # of loops required =

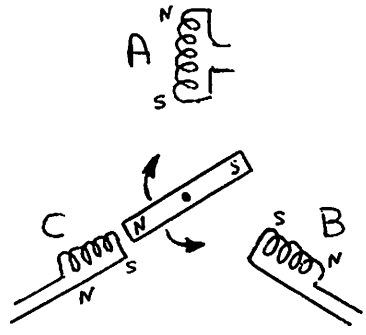
$$\frac{10}{10^3} / \frac{8}{10^6} = 1250_{10} \text{ loops}$$

Initialize the counter for this value and write the complete delay routine.

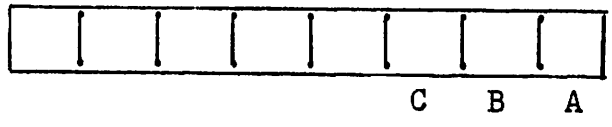
```
                    NAM     PROG68
                    OPT     O,S
                    ORG     $0100
              *
              * P8-17
              *
CE 04E2 TIMER LDX     #1250  <-- No $ sign for decimal #.
09      MORDEX DEX             4 CYCLES
26 FD          BNE    MORDEX   4 CYCLES
              *
              *                8 CYCLES TOTAL
```

This routine would then be executed when CA1 first detects a key hit, which would occur when the key is depressed, and probably upon release, which also produces transient pulses. Hence the state of CA1 should be checked after the delay. If CA1 is still 1 it is a legal key hit. If 0, it is probably due to "bounce" upon key release, which could then be ignored by the program.

A stepping motor is another application of a PIA. Imagine 3 electromagnets or coils, A, B and C, placed at equal angles around a magnet which is free to turn.



Each of electromagnets A, B and C are directly under control of a PIA Data Buffer bit, as shown in the diagram below. A magnet is ON when the appropriate bit is in the 1 state, and OFF when the bit is 0. Energizing magnet C causes the North pole of the central magnet to rotate to the South pole at C.



PIA Data Buffer B

Set up the PIA to cause the central magnet's North pole to point to A. Assume that PIABFB is already initialized for output. Also assume that the South pole of each energized electromagnet is the closest pole to the magnet, as in electromagnet C.

```
0100 86 01    MAGA    LDA A  #$01
0102 B7 7FF0          STA A  PIABFB
```

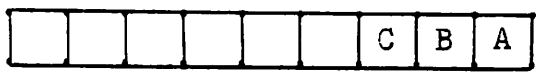
Bit #0 (electromagnet A) is ON.

How would you suggest having the N pole of the central magnet point to a half way between A and B? Write the instructions.

```
0108 86 03   MAGAB  LDA A  #$03
010A B7 7FF0        STA A  PIABFB
```

Both A and B are ON and equally attracting the N pole, causing it to point between the two electromagnets, at about the 2 o'clock position.

Write the instructions to cause the central magnet to move clockwise continuously, starting at A. Assume a delay subroutine call JSR DELAY, which introduces a delay between each change to slow down the computer changes to acceptable rotational rates.



Data Buffer

```
0100 86 01   MAGA   LDA A  #$01
0102 B7 7FF2        STA A  PIABFB
0105 BD 0132        JSR    DELAY
0108 86 03   MAGAB  LDA A  #$03
010A B7 7FF2        STA A  PIABFB
010D BD 0132        JSR    DELAY
0110 86 02   MAGB   LDA A  #$02
0112 B7 7FF2        STA A  PIABFB
0115 BD 0132        JSR    DELAY
0118 86 06   MAGBC  LDA A  #$06
011A B7 7FF2        STA A  PIABFB
011D BD 0132        JSR    DELAY
0120 86 04   MAGC   LDA A  #$04
0122 B7 7FF2        STA A  PIABFB
0125 BD 0132        JSR    DELAY
0128 86 05   MAGCA  LDA A  #$05
012A B7 7FF2        STA A  PIABFB
012D BD 0132        JSR    DELAY
0130 20 CE          BRA    MAGA
```

How would you modify the angular velocity for this stepping motor, under program control?

The constant used for the delay could be entered via a keyboard e.g., using the keys 1 - 9, each producing a different constant and therefore a different angular velocity. The smaller constant would then be down-counted sooner, producing a shorter delay, hence a higher speed.

Modern stepping motors usually have many (dozens) of coils around the circumference, alternating between A, B and C groups, each group being driven by one specific line, hence PIA bit. An output of the sequence 001, then 010, then 100 would represent one cycle, usually a few degrees. Reversing the order would reverse rotational direction.

SUBROUTINES

In previous chapters we have used subroutine calls e.g., JSR GETCHR which caused the ASCII code, for the key struck on keyboard, to appear in ACC A. Such a subroutine call causes execution of a group of instructions, headed by the label GETCHR and terminated by

RTS - ReTurn from Subroutine.

After this subroutine has been executed, the next instruction executed is that following the subroutine call, e.g.

```
JSR     GETCHR
STA A   KEYDAT
```

A program can be made up of a series of subroutine calls, each causing execution of a particular subroutine, to carry out a specific task. Each subroutine should have only one entry point and one exit point. Entry and exit conditions should be well documented in the accompanying comments, e.g., "Enter with X pointing to the head of a message, and exit when the message has been printed, with ACC A and ACC B contents being overwritten." Each subroutine can be individually tested and then used with confidence when called within the main program.

Program planning should be in "top-down" format, with overall tasks being defined first, and from these tasks the sub-tasks defined. Each task can then be assigned to a subroutine which in turn can call lower level subroutines to carry out the sub-tasks. Subroutine calls can be many levels deep, if necessary, those at the lowest level being responsible for the simplest tasks, like checking a READY bit in an ACIA or a control line in a PIA. The overall result is a hierarchical or pyramidical structure, the top levels being general or "global", the lowest levels looking after detail.

Contd...

A typical subroutine, properly documented, is shown here:

```
                * GETCHR. . SUBROUTINE WHICH RETURNS WITH
                * ASCII CHAR IN ACC A.  X AND B NOT CHANGED.
                *
      7FF4      SERCSR EQU    $7FF4
      7FF5      SERBUF EQU    $7FF5
                *
0107 B6 7FF4 GETCHR LDA A  SERCSR
010A 84 01          AND A  #$01      DATA READY?
010C 27 F9          BEQ    GETCHR    NOT YET.
010E B6 7FF5        LDA A  SERBUF    YES.  GET DATA
0111 39             RTS              AND EXIT.
```

Such a subroutine can be called from anywhere within a program, avoiding duplication of the above instructions.

A subroutine call JSR ECHO is to cause the character, struck on the keyboard, to be printed or displayed on the terminal used. ECHO itself could call 2 other subroutines. Based on this information write the subroutine ECHO, using only 3 instructions. A subroutine called PRINT is available, to print the ASCII character in ACC A.

```
                * ECHO. . SUBROUTINE TO ACCEPT ASCI CODE FROM ACIA
                * RECEIVER AND ECHO IT ON THE ACIA TRANSMITTER.
                * CALLS GETCHR AND PRINT SUBS. .
                *
0100 BD 0107 ECHO   JSR    GETCHR   GETS INPUT
0103 BD 0112        JSR    PRINT    AND OUTPUTS IT.
0106 39             RTS             AND RETURNS
```

At this point the details of GETCHR and PRINT are not necessary except that they both use ACC A.

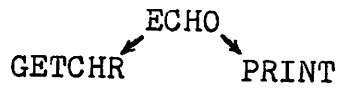
Assuming communication to the printing device via the ACIA, convert the instructions shown below to a well documented subroutine called PRINT.

```
PRINT  LDA B  SERCSR
       AND B  #$02      READY TO PRINT?
       BEQ    PRINT     NOT YET.
       STA A  SERBUF    PRINT CHAR.
```

```
               *
               * PRINT...SUBROUTINE TO PRINT ASCII CONTENTS
               * OF ACC A ON ACIA OUTPUT DEVICE.  USES A AND B.
               *
0112 F6 7FF4 PRINT  LDA B  SERCSR
0115 C4 02          AND B  #$02      READY TO PRINT?
0117 27 F9          BEQ    PRINT     NOT YET.
0119 B7 7FF5        STA A  SERBUF    PRINT CHAR.
011C 39             RTS                AND RETURN.
```

The documentation is just as important as the instructions written. Fight off the sometimes overwhelming urge to write undocumented programs, which usually end up in the waste basket, six months later.

We could depict the subroutine hierarchy as:



implying that ECHO calls both GETCHR and PRINT. For lack of a better name let's call this a "subroutine tree".

Imagine a system where the computer is to receive inputs from 2 ACIA's. It would not be feasible to have the computer wait in a loop for ACIA #1 since it could lose data from ACIA #2. The computer could alternately check ACIA #1, #2, #1 etc., receiving data from an ACIA that is ready. (The Chapter on "Interrupt" presents another solution.) A subroutine to check the READY status of ACIA #1, without reading data, is shown here.

```
0100 B6 7FF4 INCHK1 LDA A  SERCS1
0103 84 01          AND A  #$01      DATA READY?
0105 27 02          BEQ    NODATA
0107 0D             SEC               GOES HERE IF DATA READY
0108 39       SE1RTN RTS
0109 0C       NODATA CLC               GOES HERE IF NOT READY
010A 20 FC          BRA    SE1RTN
```

Upon exit from this subroutine what is different, when data is ready, compared to when data is not ready?

The C bit is set when data is ready, and cleared when data is not ready.

In Appendix C find 2 instructions, each of which branch conditionally, depending on the state of the C bit. Use one of them in the main program below, upon return from the subroutine INCHK1 to determine whether or not to store data, MEMAD1 being the pointer. If that is not too difficult repeat for ACIA #2, where MEMAD2 is the pointer within INCHK2, which similarly checks if ACIA #2 is ready.

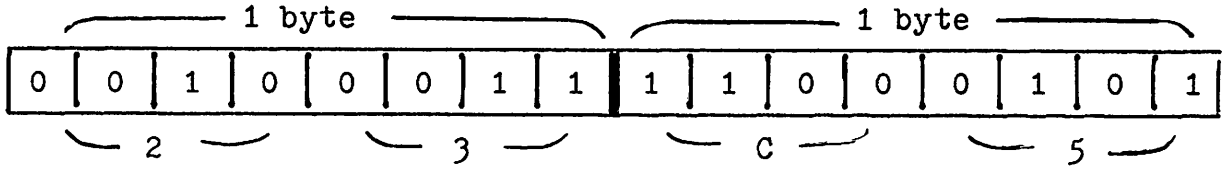
BCC - Branch if Carry bit Cleared

or BCS - Branch if Carry bit Set

```
CHECK1  JSR     INCHK1   ACIA #1 READY?
        BCC     CHECK2   NO DATA HERE
        LDX     MEMAD1
        INX
        STX     MEMAD1   GET POINTER
        LDA A   SERBF1   GET INPUT DATA
        STA A   X        AND STORE IT.
CHECK2  JSR     INCHK2   ACIA #2 READY?
        BCC     CHECK1   NO DATA HERE
        LDX     MEMAD2
        INX
        STX     MEMAD2   GET POINTER
        LDA A   SERBF2   GET DATA FROM #2
        STA A   X        AND STORE IT.
        BRA     CHECK1
```

The use of the C bit permits decisions to be made within a subroutine, without violation of the requirement for a single return to the mainline program, via <u>one</u> RTS instruction. The RTS should be the <u>only</u> means of exiting from a subroutine. To violate this rule, e.g., via a branch instruction, destroys the modular design of your program and makes de-bugging a nightmare.

Let's look at a subroutine HEXADD which expects 4 hex keys to be struck, and stores the corresponding 4 character hex value in 2 consecutive bytes of memory. For example if keys 2, 3, C and 5 are struck, the 2 bytes of memory would look like this:



Approaching this from a "top-down" direction, assume that we have a subroutine INBYTE which would return with 23_{16} in ACC A when two keys, 2 and 3, are struck. Write the subroutine HEXADD which calls INBYTE and produces the 16 bit binary contents in the two memory locations, ADDRES and ADDRES+1.

```
                * HEXADD...STORES 2 BYTES IN MEM AT LABEL ADDRES
                * CALLS INBYTE TWICE.  USES ACC A.
                *
0100 BD 0113 HEXADD JSR     INBYTE    GET 8 BITS IN ACC A.
0103 B7 010D        STA A   ADDRES    AND STORE THEM.
0106 BD 0113        JSR     INBYTE    8 MORE BITS
0109 B7 010E        STA A   ADDRES+1  INTO NEXT ADDRESS.
010C 39             RTS
010D 0002    ADDRES RMB     2
                *
     0113    INBYTE EQU     $0113
```

This "top-down" approach assumes that we could write the INBYTE subroutine, if it is not already available.

Now also assume that INBYTE returns with the C bit set if an invalid hex key was struck; otherwise C is cleared. Modify the HEXADD subroutine to check for this abnormal condition, restarting the HEXADD subroutine when such an error is detected. Modify the documentation accordingly.

```
                    *
                    * HEXADD. . . STORES 2 BYTES IN MEM AT LABEL ADDRES
                    * CALLS INBYTE TWICE, CHECKING FOR ERROR WITHIN BYTE
                    * SUB VIA SET C BIT.  ACC A USED.
                    *
                    *
0100 BD 0113 HEXADD JSR     INBYTE    GET 8 BITS IN ACC A.
0103 25 FB          BCS     HEXADD    RESTART IF ERROR.
0105 B7 0111        STA A   ADDRES    ELSE STORE THEM.
0108 BD 0113        JSR     INBYTE    8 MORE BITS
010B 25 F3          BCS     HEXADD    RESTART IF ERROR .
010D B7 0112        STA A   ADDRES+1  ELSE STORE IN NEXT ADDRESS.
0110 39             RTS
0111 0002    ADDRES RMB     2
```

A better solution would be to print the message BAD HEX before restarting HEXADD. This improves communcation between the computer and the user, an important consideration in program design.

9-7

A subroutine HEXCHR is now available to acquire an ASCII character in ACC A, when a key is struck, and to convert it to its 4 bit hex equivalent, e.g., 0B results when B is struck. This 4 bit result will be right-justified (against the right edge or as far right as possible) in ACC A. Is this where you ultimately want the first 4 bits inside ACC A when the INBYTE subroutine, which receives two such characters, is executed?

No. If 5 is the first of two keys struck, the 0101 result must be moved to the left half of ACC A, to make room for the next 4 bits, which go in the right half when the second key is struck.

9-8

Write the first half of the INBYTE subroutine to place the first 4 bits in the left half of ACC B. Useful instructions might be ASL A and TAB. Why is ACC B needed? The HEXCHR subroutine is still available and returns with the C bit set if an invalid hex key was struck. Such a condition should cause an immediate return from INBYTE to HEXADD, with the C bit still set.

```
0113 BD 0125 INBYTE JSR    HEXCHR   GET 4 BITS
0116 25 0C          BCS    BYTRTN   BAD HEX. RETURN NOW.
0118 48             ASL A
0119 48             ASL A
011A 48             ASL A
011B 48             ASL A           SHIFT 4 BITS LEFT.
011C 16             TAB             STORE IN B
```

ACC B is used to store the first 4 bits when HEXCHR, which uses ACC A, is called to get the second 4 bits. RTS passes the C bit, undisturbed, to the calling subroutine HEXADD.

Now finish the INBYTE subroutine including documentation. The instruction ABA may be useful to you.

The complete INBYTE subroutine might be:

```
                * INBYTE... PRODUCES 8 BITS IN ACC A CORRESPONDING
                * TO TWO 4 BIT HEX VALUES, EACH PRODUCED BY
                * HEXCHR SUB, WHICH IS CALLED TWICE. USES A AND B
0113 BD 0125 INBYTE JSR    HEXCHR   GET 4 BITS
0116 25 0C          BCS    BYTRTN   BAD HEX. RETURN NOW.
0118 48             ASL A
0119 48             ASL A
011A 48             ASL A
011B 48             ASL A           SHIFT 4 BITS LEFT.
011C 16             TAB             STORE IN B
011D BD 0125        JSR    HEXCHR   GET 4 MORE BITS.
0120 25 02          BCS    BYTRTN   IF BAD HEX
0122 1B             ABA             MERGE BOTH 4 BIT SETS OF DATA
0123 0C             CLC             TELL THEM ITS GOOD DATA
0124 39      BYTRTN RTS
```

ACC A	0 0 0 0 1 1 1 0	-- After the first JSR HEXCHR if E was struck.
ACC B	1 1 1 0 0 0 0 0	-- After the TAB instruction.
ACC A	0 0 0 0 1 0 0 1	-- After the second JSR HEXCHR if 9 was struck.
ACC A	1 1 1 0 1 0 0 1	-- After ABA. ACC B is added to ACC A to merge both 4 bit codes.

So far we have HEXADD calling INBYTE twice.

The HEXCHR subroutine could be formed from the hex checking program shown early in the Branching Chapter. Write this subroutine including the following changes:

(a) At the beginning of the subroutine get the ASCII code for the struck key into ACC A.

(b) Set the C bit if an invalid hex key is struck; otherwise clear the C bit and return from the subroutine with the 4 bit hex code in ACC A.

Refer to the Branching Chapter for the original hex checking program. Assume that the GETCHR subroutine is available to receive an ASCII code in ACC A, when a key is struck.

```
                    * HEXCHR...RECEIVES ASCII CODE IN ACC A VIA GETCHR
                    * CONVERTS TO 4 BIT HEX EQUIVALENT IF VALID
                    * AND CLEARS C BIT.  ELSE RETURNS WITH C SET.
                    *
0125 BD 0143 HEXCHR JSR     GETCHR    (ECHO WOULD BE BETTER STILL)
0128 81 2F          CMP A   #$2F
012A 23 14          BLS     BADHEX    BELOW 30, NOT HEX
012C 81 39          CMP A   #$39
012E 23 0C          BLS     NUMOK     0 TO 9.  VALID HEX
0130 81 40          CMP A   #$40
0132 23 0C          BLS     BADHEX    3A TO 40.  ILLEGAL
0134 81 46          CMP A   #$46
0136 22 08          BHI     BADHEX    ABOVE 46.  ILLEGAL
0138 80 37          SUB A   #$37      A TO F IN 4 BIT FORMAT
013A 0C       GOODHX CLC              TELL THEM IT'S GOOD
013B 39       HEXRTN RTS
013C 80 30    NUMOK  SUB A  #$30      0 TO 9 IN 4 BIT FORMAT.
013E 20 FA           BRA    GOODHX
0140 0D       BADHEX SEC              BAD NEWS.  WRONG KEY.
0141 20 F8           BRA    HEXRTN
```

The GETCHR subroutine is essentially the same as before except for 2 changes:

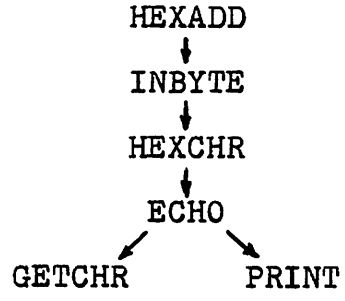
(a) Bit #7, the parity bit must be cleared for all data.

(b) Lower case alphabetic characters a to z, must be forced to upper case by clearing bit #5. Write the GETCHR subroutine.

Both of the above are required to make the data independent of the type of terminal (some produce parity bit set, others cleared) and to eliminate having to hold the SHIFT key down when entering alphabetic characters.

```
                * GETCHR...SUBROUTINE TO GET ASCII CODE FROM ACIA RX.
                * BIT #7 (PARITY BIT) CLEARED. UPPER CASE IS FORCED.
                *
     7FF4       SERCSR EQU    $7FF4
     7FF5       SERBUF EQU    $7FF5
                *
0143 B6 7FF4    GETCHR LDA A  SERCSR
0146 84 01             AND A  #$01      DATA READY?
0148 27 F9             BEQ    GETCHR    NOT YET.
014A B6 7FF5           LDA A  SERBUF    YES. GET DATA
014D 84 7F             AND A  #$7F      CLEAR PARITY BIT.
014F 81 60             CMP A  #$60
0151 23 06             BLS    GETRTN    BELOW "SMALL A"
0153 81 7A             CMP A  #$7A
0155 22 02             BHI    GETRTN    ABOVE "SMALL Z"
0157 84 DF             AND A  #$DF      UPPER CASE ALPHA CHAR
0159 39         GETRTN RTS               AND EXIT.
```

Describe the sequence of events when a non-hex key is struck. Sketch the "subroutine tree" in your answer.



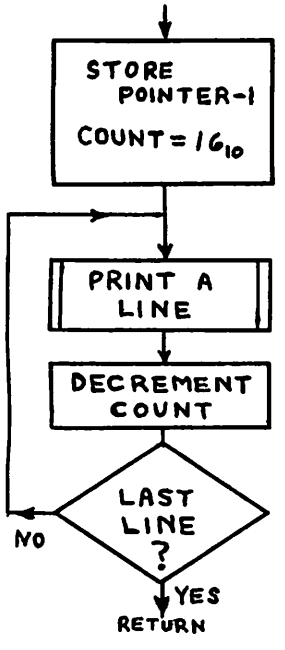
When HEXCHR detects an invalid hex character the C bit is set and HEXCHR returns to INBYTE. INBYTE immediately checks the C bit and, noting that the C bit is set, returns immediately to HEXADD, which also checks the C bit. HEXADD, on noting that the C bit is set, immediately restarts. In summary, a wrong key immediately restarts HEXADD, preferably after a printed message such as BAD HEX.

Further use of the C bit is seen in a program where a task, assigned to a subroutine, results in the C bit being cleared if the task is completed normally. If the result is abnormal the C bit is set and ACC A contains the erroneous result, which can be printed as an error message.

Here is a new problem, to write a subroutine called PAGE which prints one page of data, the first address of the data being in the X Register when PAGE is called. The format is as follows:

- one PAGE comprises 16_{10} lines.
- one LINE comprises a Carriage Return and Line Feed (to start a new line) followed by 8 words, each separated by a space.
- one WORD comprises 4 bytes, from memory, each byte being printed as 2 ASCII characters, e.g., 00111101 in memory would cause 3D to be printed.

Use a "top-down" approach to this problem in flow charting and writing the subroutine PAGE. Assume that the subroutine LINE is available to print one LINE.



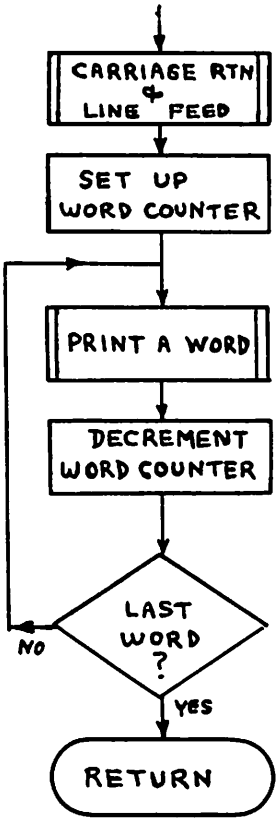

```
* PAGE...SUBROUTINE TO PRINT ONE PAGE (16 LINES)
* OF DATA FROM MEMORY.  ENTER WITH X POINTING TO
* FIRST CHAR TO BE PRINTED.  CALLS LINE SUB.
*
PAGE    DEX                 ONE BELOW FIRST CHAR ADDR
        STX     MEMPNT      INIT POINTER.
        LDA A   #16
        STA A   LINCNT      SET UP COUNTER.
NULINE  JSR     LINE        PRINT LINE
        DEC     LINCNT      LAST LINE?
        BNE     NULINE      NO.  PRINT ANOTHER
        RTS                 LAST ONE.
LINCNT  RMB     1
MEMPNT  RMB     2
```

Note the double vertical bar here indicating a subroutine.

The address for the first memory address could be produced by the previous subroutine HEXADD.

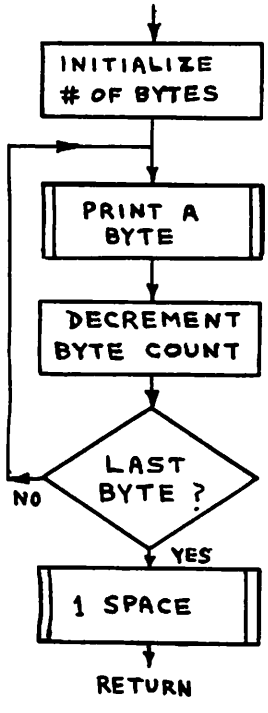
The next task, working downward, is to write the subroutine LINE, which prints 8 words, each comprising the contents of 4 addresses. Flow chart and write the subroutine LINE, assuming that 2 subroutines are available as follows:

- WORD, to print one word.
- CRLF, to produce a Carriage Return and Line Feed, to start the next character on a new line.



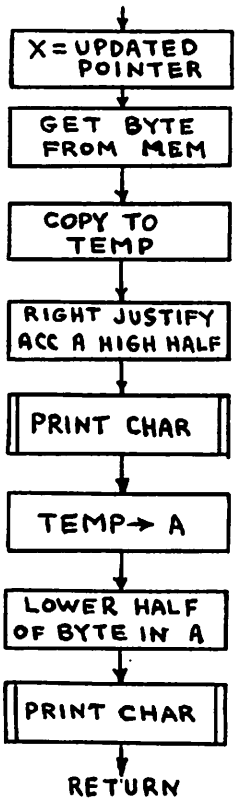

```
* LINE...SUBROUTINE TO PRINT 64 (DECIMAL) CHAR
* FROM 32 MEMORY ADDRESSES. CALLS WORD. USES A.
*
LINE    JSR     CRLF        START NEW LINE
        LDA A   #$08
        STA A   WRDNUM      SET UP COUNTER
NUWORD  JSR     WORD
        DEC     WRDNUM      LAST WORD?
        BNE     NUWORD      NO. BACK AGAIN.
        RTS                 LAST ONE.
WRDNUM  RMB     1
*
```

The next subroutine proceeding downward is WORD, which prints the contents of 4 memory locations, then skips one space. The subroutine OBYTE, to print the contents of ACC A as 2 ASCII characters is available. SPACE, another subroutine will print (or skip over) one space. Flow chart and write the WORD subroutine.




```
* WORD...SUBROUTINE TO PRINT CONTENTS OF 4 MEM
* ADRESSES AS 8 HEX CHAR.  CALLS OBYTE AND SPACE.
* USES ACC A.
*
WORD    LDA A   #$04
        STA A   BYTCNT      INIT COUNTER
NUBYTE  JSR     OBYTE       PRINT 1 BYTE AS 2 CHAR.
        DEC     BYTCNT
        BNE     NUBYTE      NOT LAST BYTE
        JSR     SPACE       YES. LAST BYTE.  ONE SPACE.
        RTS                 DONE
BYTCNT  RMB     1
```

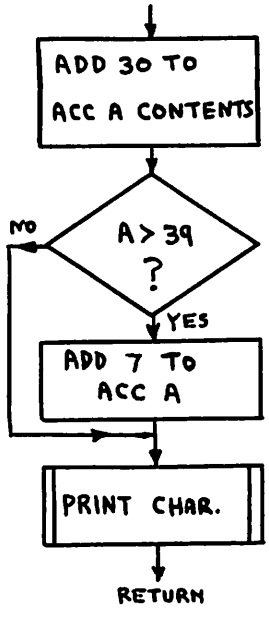
The OBYTE subroutine is next. It gets one byte from memory via the pointer MEMPNT and calls HEXPRT twice to print it as 2 ASCII characters. HEXPRT is entered with 4 bits right-justified in ACC A. Flow chart and write the OBYTE subroutine.




```
* OBYTE. . . SUBROUTINE TO PRINT CONTENTS OF ONE MEM
* ADDRESS AS 2 ASCII CHAR.  CALLS HEXPRT.  USES A ,X.
* ENTER WITH ADDRESS IN MEMPNT.
*
OBYTE   LDX     MEMPNT
        INX
        STX     MEMPNT      GET ADDRESS
        LDA A   X           GET BYTE
        STA A   TEMP        SAVE COPY.
        ASR A               RIGHT
        ASR A               JUSTIFY
        ASR A               LEFT
        ASR A               HALF
        AND A   #$0F        ZAP LEFT HALF
        JSR     HEXPRT      PRINT IT
        LDA A   TEMP        GET CLEAN COPY
        AND A   #$0F        ZAP LEFT HALF
        JSR     HEXPRT      PRINT IT
        RTS                 DONE
TEMP    RMB     1
```

Note the use of TEMP rather than ACC B. It is not good practice to tie up an accumulator, when calling a subroutine which may need the accumulator.

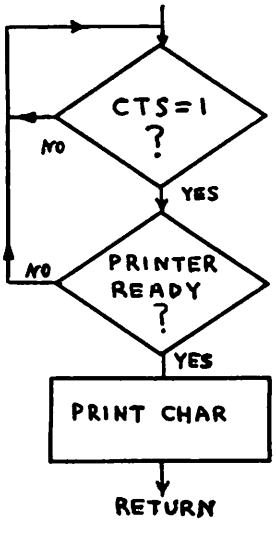
HEXPRT is entered with 4 bits right-justified in ACC A. It prints the corresponding ASCII character. Flow chart and write this subroutine noting that PRINT is available to print the ASCII contents of ACC A.




```
* HEXPRT...SUB TO PRINT ASCII CHAR. CALLS PRINT SUB.
* ENTER WITH 4 BITS RIGHT JUSTIFIED IN ACC A.
*
HEXPRT ADD A  #$30     CONVERT TO ASCII
       CMP A  #$39     NUMBER?
       BLS    OUTPUT   YES.
       ADD A  #$07     LETTER. ADD 7 MORE.
OUTPUT JSR    PRINT    OUT IT GOES.
       RTS
```

Check this routine by testing it first with values 0 and 9, then with values A and F, plus the 4 values just outside these legal values.

Next we need the PRINT subroutine. The printer, via the $\overline{CTS}$ control line back to the ACIA, will inform the computer to stop transmitting while Carriage Return and Line Feed functions take place. Flow chart and write the subroutine to transmit data via the ACIA when CTS = 1 ($\overline{CTS}$ = 0).



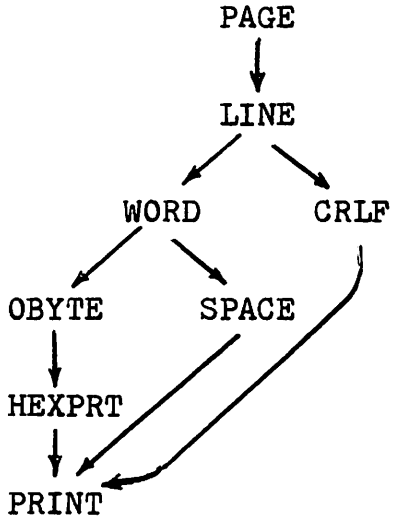

```
* PRINT...SUBROUTINE TO PRINT CHAR IF DEVICE
* IS ON LINE VIA CTS=1 (CTS NOT=0). USES ACC A AND B
* ENTER WITH ASSCII CODE IN ACC A.
SERCSR EQU    $7FF4
SERBUF EQU    $7FF5
*
PRINT  LDA B  SERCSR
       BIT B  #$08      CTS NOT=0?
       BNE    PRINT     NO. TRY AGAIN.
       BIT B  #$02      READY?
       BEQ    PRINT
       STA A  SERBUF    PRINT IT
       RTS
```

Loopback for the second test is to the top to ensure that $\overline{CTS}$ has not gone to 1, while waiting for the printer to become READY.

SPACE and CRLF now remain. A problem exists in using the ACIA with the printer in that the ACIA will transmit the last character in its TRANSMIT Buffer even though the printer requests a halt to more data by clearing $\overline{CTS}$ (Clear To Send). $\overline{CTS}$ is normally cleared during a Carriage Return or Line Feed operation or when the printer is not ready to print data. The above problem results in the loss of the last transmitted character. The solution is to send a 2 nulls (0C) to the ACIA after both the CR and LF characters. The nulls are then "sacrificed" to preserve the next legal character printed. With this in mind, write the CRLF and SPACE subroutines. Flow charts are not necessary for these.

```
* SPACE. . . SUBROUTINE TO OUTPUT ONE SPACE CHAR.
* CALLS PRINT SUB.  USES ACC A.
*
SPACE   LDA A   #$20      ASCII FOR SPACE
        JSR     PRINT
        RTS

* CRLF. . . SUBROUTINE TO OUTPUT CARRIAGE RETURN
* AND LINE FEED CHAR TO PRINTING DEVICE.  PADS EACH
* WITH 2 NULLS CHAR.  CALLS PRINT SUB.  USES ACC A.
*
CRLF    LDA A   #$0D      CR
        JSR     PRINT
        CLR A
        JSR     PRINT     OUTPUT NULL
        JSR     PRINT
        LDA A   #$0A      LF
        JSR     PRINT
        CLR A
        JSR     PRINT     NULL
        JSR     PRINT
        RTS
```

To complete the subroutine PAGE, draw the "subroutine tree" to show the subroutine's hierarchy.

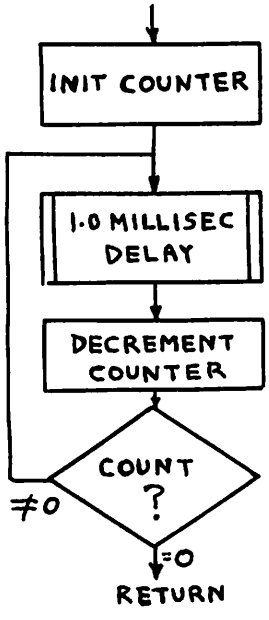


In only a few words, the overview of PAGE is depicted here.

A program could call both the HEXADD and PAGE subroutines, the former to define the starting address and the latter to print the page of data.

Near the end of the PIA chapter is a program in which a delay is used to "de-bounce" a switch before its state is read by the PIA. This delay could be achieved more easily if subroutine format was used.

Flow chart and write a subroutine which produces a delay of N milliseconds, where N is the binary contents of ACC A. This subroutine should call a subroutine MILSEC which produces a delay of 1 millisecond each time it is called. Write the MILSEC subroutine, assuming 1 microsecond per MPU cycle. If necessary refer to the PIA chapter for the previous delay routine.




```
                    * DELAY...SUBROUTINE TO PRODUCE DELAY
                    * OF N MILLISECONDS, WHERE N= BINARY
                    * CONTENTS OF ACC A ON ENTRY.  CALLS MILSEC.
                    *
0100 B7 010C DELAY  STA A  COUNT    STORES N
0103 BD 010D MORMIL JSR    MILSEC   ONE MILLISEC
0106 7A 010C        DEC    COUNT
0109 26 F8          BNE    MORMIL   NOT LAST YET
010B 39             RTS
010C 0001    COUNT  RMB    1
                    *
                    * MILSEC...SUB TO PROVIDE
                    * ONE MILLISECOND DELAY.
                    *
010D 86 64   MILSEC LDA A  #100
010F B7 0118        STA A  MILCNT
0112 7A 0118 MORDEC DEC    MILCNT
0115 26 FB          BNE    MORDEC
0117 39             RTS
0118 0001    MILCNT RMB    1
```

The 2 loop instructions DEC MILCNT and BNE MORDEC take $6 + 4 = 10_{10}$ MPU cycles or 10 microseconds. Therefore 100_{10} or 64_{16} loops provide a delay of 1000 microseconds or one millisecond.

In the previous frame MILCNT could have been given an initial value of 64_{16} simply via

```
MILCNT FCB $64
```

eliminating the need for the 2 lines of initialization at the start of the MILSEC subroutine. Would this be acceptable? Why?

No! The subroutine would execute properly the first time it is called, MILCNT being decremented from 64 to 0. The second time (and all subsequent times) that it is called MILCNT would start at FF, after first being decremented from 0 by DEC MILCNT. This subroutine MILSEC would then go through 256_{10} loops to reach zero, instead of 100_{10} loops, producing an incorrect delay. Self-initialization is required within the subroutine to reset MILCNT to 64 <u>every</u> time the subroutine is called. Lack of self-initialization is a common catastrophic error when coverting a program, which runs correctly once, into a subroutine which is called many times within a larger program.

This concept should be extended to all programs, as well as subroutines enabling faulty programs to be restarted during de-bugging without the necessity of being reassembled or reloaded.

Enough said for now about subroutines!

STACK OPERATIONS

Previously we have seen data storage in which the Index Register was used as a pointer. Another 16 bit register, the Stack Pointer (SP) is also used to store and retrieve data, employing a user-defined block of memory, called the stack, for the storage operations. The Stack Pointer may be initialized to point to the address 1C40 via

LDS #$1C40 (LoaD the Stack pointer)

Another instruction

PSH A (PuSH accumulator A)

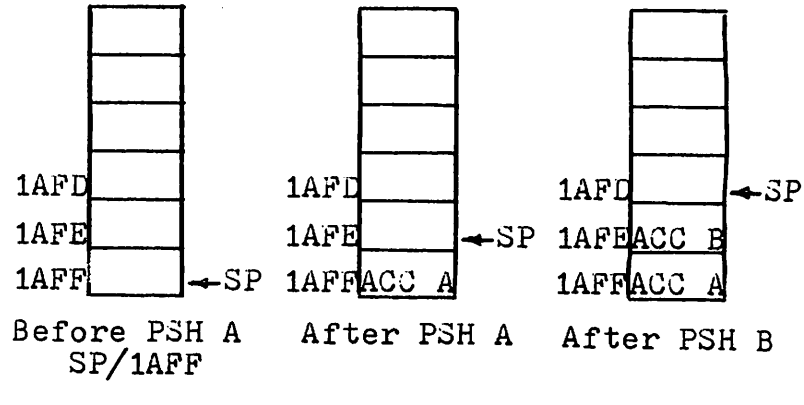
performs a "push" operation, that is it stores the contents of ACC A in the address now contained in the Stack Pointer. The Stack Pointer is automatically decremented after the storage operation.

"PuSH" is an appropriate description, similar to the "pushing" of individual serviettes into a metal holder, each new serviette now being on the top of the stack.

Initialize the Stack Pointer to 1AFF, then store the contents of ACC A and ACC B on the stack in that order.

```
0100 8E 1AFF        LDS    #$1AFF
0103 36             PSH A
0104 37             PSH B
```

Stack Status Diagrams



Data can be retrived from the top of the stack via

PUL A

which "pulls" the data off the stack into ACC A. This is similar to retrieving a stored serviette from the holder, the last one in being the first one out. In the PUL operation the stack pointer is incremented automatically, before each byte is retrieved. Assuming the 2 PSH operations in the previous frame the instructions:

```
33          PUL B
32          PUL A
```

first transfers the data, stored in 1AFE, into ACC B, then transfers the data from 1AFF into ACC A. Note that the PUL operations are in the reverse order to the PSH operations, respecting the "Last In First Out" (LIFO) sequence.

Use of the stack permits temporary storage of data without the need for a symbolic address or an accumulator usage. Modify this now familiar subroutine to operate without ACC B. Assume previous stack pointer initialization.

```
PRINT  LDA B  SERCSR
       AND B  #$02      READY TO PRINT?
       BEQ    PRINT     NOT YET.
       STA A  SERBUF    PRINT CHAR.
       RTS              AND RETURN.
```

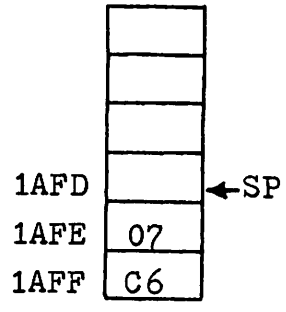
```
     7FF4     SERCSR EQU    $7FF4
     7FF5     SERBUF EQU    $7FF5
              *
0100 36       PRINT  PSH A
0101 B6 7FF4  NOTYET LDA A  SERCSR
0104 84 02           AND A  #$02
0106 27 F9           BEQ    NOTYET
0108 32              PUL A
0109 B7 7FF5         STA A  SERBUF
010C 39              RTS
```

WARNING: For every PSH there must be a corresponding PUL to restore the stack pointer to its original state.

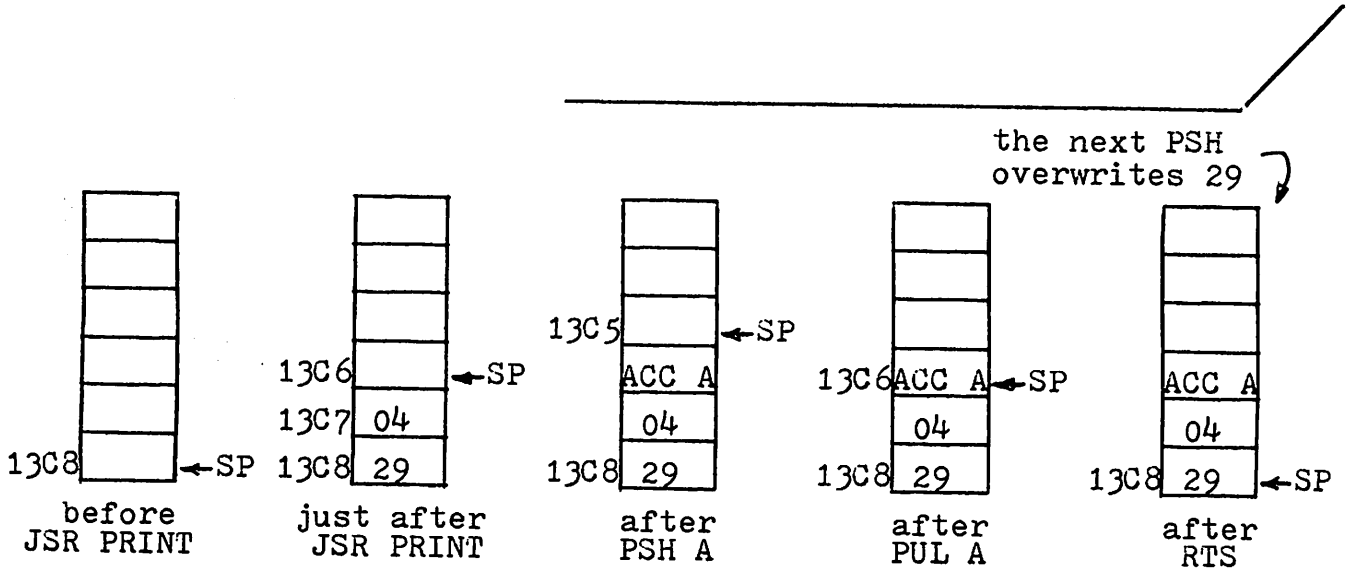
Assume that the main line program which calls this PRINT subroutine is:

```
07C3 BD 1358        JSR    PRINT
07C6 FE 077E        LDX    MEMPNT
```

If the stack pointer contains 1AFF just before JSR PRINT is executed, the address of the next main line instruction, 07C6 in this example, is stored on the stack. The low byte (C6) goes into 1AFF and the high byte (07) goes into 1AFE. The stack status at this point is depicted by this diagram. The RTS instruction at the end of the subroutine automatically performs two PUL operations, restoring the 07C6 value in the Program Counter. The next instruction executed is then from 07C6, the LDX MEMPNT instruction following the subroutine call.



Assume that the first byte of JSR PRINT resides in 0426, and that the stack pointer contents is 13C8 just before JSR PRINT is executed. Draw the stack diagram showing stack contents and SP value for each stack change, starting just before JSR PRINT is executed and finishing when LDX MEMPNT is executed. The PRINT subroutine is the one given in the answer of the previous frame.



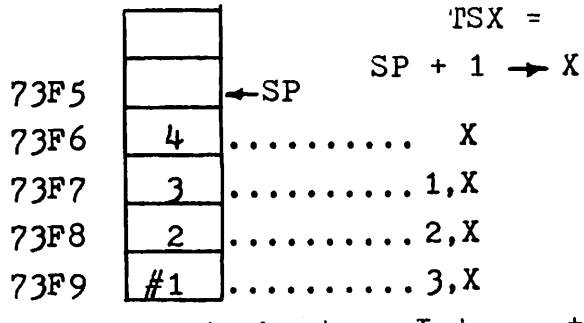
Examination of data stored on the stack is achieved via:
TSX - Transfer Stack pointer to indeX register.

which transfers the Stack Pointer to the Index Register, then increments the Index Register. In this way the Index Register points at the last byte stored on the stack. This permits direct access to the data, stored on the stack, without disturbing the Stack Pointer. Write the instructions to print the value of the last byte, stored on the stack. The subroutine OBYTE is available.

```
0203 30          TSX
0204 A6 00       LDA A  X
0206 BD 0142     JSR    OBYTE
```


Assume that 4 bytes have been stored on the stack. It is now desired to increment the first of these 4 bytes without disturbing the stack pointer or other data on the stack. Write the necessary instructions.

```
0100 30          TSX
0101 6C 03       INC    3,X
```



More stack operations will be seen in the next chapter, Interrupt, where the stack is used extensively.

INTERRUPT

The simplest type of "interrupt" operation is that produced when you start the 6800 microcomputer by pushing the RESET button. This starts execution of a permanently stored program or "service routine", as interrupt initiated programs are called, this one servicing the RESET button. When this button is pushed the $\overline{\text{RESET}}$ line to the MPU is grounded. This causes the computer to look in addresses FFFE and FFFF (called "vector" addresses) for the address of the RESET service routine. The RESET service routine is then started, typically clearing all READY bits, initializing the stack pointer and setting up input/output devices such as the PIA or ACIA for the required mode of operation.

The $\overline{\text{RESET}}$ line also can be converted to force a restart of this service routine automatically when power is first applied, eliminating the RESET button. This is particularly useful when the microcomputer controls an electronic subsystem or an appliance (e.g., microwave oven).

Another form of interrupt provides the solution to the problem of determining when a peripheral device has data or requires data, without the continuous check of READY bits in an ACIA or PIA. Under interrupt operation, such devices are ignored by the computer until the device demands service, whereupon the computer suspends its present operation, known as a "background" program and executes the service routine or "foreground" program for the device which demanded service.

Such service may involve the transfer of one byte of data or the change of several bits in a status register. When the service routine is completed the computer resumes execution of the background program.

Several points are relevant to interrupt operations:

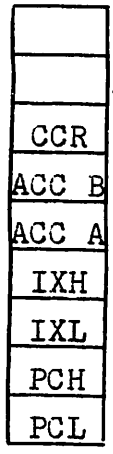
(a) As stated above, READY bit polling or testing, as a routine operation, is now eliminated permitting more flexible and efficient use of the computer. With interrupt operation the peripheral devices essentially say to the computer "Don't call us. We'll call you."

(b) The service routine is entered each time that a character is transmitted or received by the interrupting device or each time that a push button activates a PIA Control Line. Such a service routine is short, typically requiring 30 to 60 microseconds to execute.

(c) The elapsed time between successive interrupts by a particular device is usually long, compared to the execution time for a service routine. Even at high data rates such as 960 characters/sec., the time between successive interrupts is approximately 1 millisecond. For push button activated interrupts this time could be seconds to hours. Consequently it is possible to service many devices via interrupt and still execute background programs for a large percentage of the computer's available time.

(d) Interrupt programs are not recommended initially because programming errors are more difficult to find. Orderly de-bugging, possible with nested subroutine type programs, is less applicable here because the occurrence of interrupts is essentially random in time. This makes it difficult to determine the conditions of various registers at interrupt time, if a service routine occasionally fails.

Interrupt servicing of interfaces such as the ACIA or PIA usually involves "Interrupt ReQuest" or "IRQ" operation, also known as "Maskable Interrupt". Such an interrupt request is made by grounding of the $\overline{IRQ}$ line to the MPU by the interrupting interface. This causes the present contents of the Program Counter, Index Register, ACC A, ACC B and the CCR to be pushed automatically on the stack in the above order. After providing service to the interrupting device the IRQ service routine is terminated by the instruction



RTI (ReTurn from Interrupt)

which automatically pulls the stored values from the stack, restoring the above registers and accumulators to their state when IRQ operation was requested. Resumption of the background program takes place as if nothing happened (except for the slight delay to provide IRQ service).

IRQ operation first requires initialization of the IRQ Vector Addresses, FFF8 and FFF9, with the address of the IRQ Service Routine. IRQ operation (interrupt service) will then take place if all the following are true:

(a) The Control Register of the appropriate interface (ACIA or PIA) has been <u>permitted</u> to interrupt. For example bit #7 of the ACIA Control Register is set to permit ACIA Receiver Interrupt. PIA interrupt via CA1 is permitted by setting Control Register bit #0.

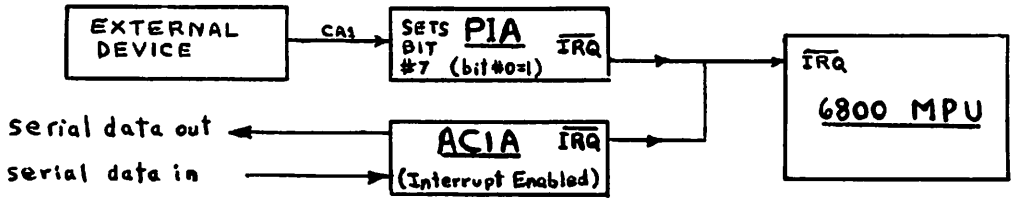
(b) The interface (ACIA or PIA) must activate (ground) the $\overline{IRQ}$ line. This happens automatically when the READY bit is set, indicating that data is ready from the ACIA Receiver, or that data is needed by the ACIA Transmitter, or that an input Control Line in the PIA is now ACTIVE.

(c) The I (Interrupt) bit of the CCR must be cleared, e.g., via the instruction

CLI (CLear Interrupt)

which permits all IRQ-connected interfaces to interrupt. Hence IRQ operation is controlled "globally" via the I bit and locally via each Control Register.

The PIA and ACIA, connected for interrupt operation, are shown in the block diagram below.



Before the I bit is cleared to permit IRQ operation, several preparations for interrupt operation must be made, usually referred to as "background initialization". These are:

(a) Set up the IRQ vector addresses FFF8 and FFF9 with the service routine address.

(b) Set the Control Register bits of the appropriate interface (ACIA or PIA) to permit an IRQ request via the receiver, transmitter or Control Line.

(c) Set up any data pointers for storing or retrieving data.

Only now can the I bit be cleared to permit IRQ operation.

Write the background initialization to set the address of ACIARX, the start of the ACIA service routine, in addresses FFF8 and FFF9.

```
LDX     #ACIARX
STX     $FFF8     INIT VECTOR FOR IRQ
```

When an interrupt occurs, the contents of the accumulators and registers will be pushed on the stack. Then the address of the next instruction to be executed will be obtained from FFF8 and FFF9, the IRQ vector address. In other words the next instruction to be executed will be the first instruction of the the IRQ service routine.

Continuing with the background initialization, set the ACIA Receiver Interrupt bit, to permit interrupt to occur. Then initialize MEMADD with the address one below address 1A00, to permit storage of data from the ACIA Receiver. Assume, as before, that ACIACR is the "original" for the "write only" Control Register of the ACIA.

```
LDA A  ACIACR
ORA A  #%10000000  ENABLE RX INT
STA A  ACIACR
STA A  SERCSR
LDX    #$1A00-1
STX    MEMADD   SET UP STORAGE POINTER.
```

So far the background initialization is:

```
0100 CE 011C        LDX    #ACIARX
0103 FF FFF8        STX    $FFF8    INIT VECTOR FOR IRQ
0106 B6 738E        LDA A  ACIACR
0109 8A 80          ORA A  #%10000000  ENABLE RX INT
010B B7 738E        STA A  ACIACR
010E B7 7FF4        STA A  SERCSR
0111 CE 19FF        LDX    #$1A00-1
0114 FF 011A        STX    MEMADD   SET UP STORAGE POINTER.
```

Now complete the background initialization by clearing the interrupt bit in the Condition Code Register. At this point a background task could be started. Since we have no background task to do at this time, put the computer in an endless loop, which will be interrupted from time to time by the ACIA, when it receives another character.

```
        CLI             ENABLE INTERRUPT
HR      BRA     HR      BACKGROUND LOOP
```

The complete background initialization to provide interrupt service for the ACIA Receiver is then

```
0100 CE 011C        LDX     #ACIARX
0103 FF FFF8        STX     $FFF8     INIT VECTOR FOR IRQ
0106 B6 738E        LDA A   ACIACR
0109 8A 80          ORA A   #%10000000  ENABLE RX INT
010B B7 738E        STA A   ACIACR
010E B7 7FF4        STA A   SERCSR
0111 CE 19FF        LDX     #$1A00-1
0114 FF 011A        STX     MEMADD    SET UP STORAGE POINTER.
0117 0E             CLI               ENABLE INTERRUPT
0118 20 FE   HR     BRA     HR        BACKGROUND LOOP
011A 0002    MEMADD RMB     2
```

Now write the service routine ACIARX, which stores one byte via MEMADD each time that the service routine is entered. Terminate this service routine with RTI, which returns control to the interrupted background program.

```
                * INTERRUPT SERVICE ROUTINE FOR ACIA RX.
                * STORES ONE CHAR IN MEM VIA MEMADD POINTER.
                *
011C FE 011A ACIARX LDX     MEMADD
011F 08             INX
0120 FF 011A        STX     MEMADD    GET NEXT ADDRESS
0123 B6 7FF5        LDA A   SERBUF    GET DATA
0126 A7 00          STA A   X         AND STORE VIA MEMADD
0128 3B             RTI               AND RETURN TO BACKGROUND.
```

Each time that the ACIA's Receiver is READY with another byte of data, bit #0 of its Status Register will go to 1, indicating the READY condition. Since bit #7 of the ACIA Control Register is also set, permitting ACIA Receiver Interrupt, the setting of the READY bit automatically activates the $\overline{IRQ}$ line to the MPU, causing execution of the service routine whose starting address is in FFF8 and FFF9. After the RTI instruction of this service routine the background task, if there is one, will be resumed. A long story isn't it?

Printing a message via the ACIA under interrupt is similar to data reception in the previous frame. Here the ACIA Control Register bits #6 and 5 must be initialized to provide "$\overline{RTS}$ = low, Transmitting Interrupt Enabled". (See Appendix E).

Write the background initialization to permit printing of the message INVALID HEX via the ACIA under interrupt. Include the message in the background initialization.

```
0100 CE 012A          LDX     #MESPRT  GET INT ROUTINE ADDRESS
0103 FF FFF8          STX     IRQVEC   INIT MESSAGE POINTER
0106 B6 738E          LDA A   ACIACR
0109 84 BF            AND A   #%10111111  CLEAR BIT 6
010B 8A 20            ORA A   #%00100000  SET BIT 5 TX INT ENABLED
010D B7 738E          STA A   ACIACR   UPDATE ORIGINAL
0110 B7 7FF4          STA A   SERCSR   SET UP ACIA
0113 CE 011B          LDX     #BADHEX-1
0116 FF 0128          STX     MEMADD   SET UP POINTER
0119 0E               CLI
011A 20 FE    HR      BRA     HR       SPIN FOREVER
              *
011C 49       BADHEX  FCC     /INVALID HEX/
0127 00               FCB     0
0128 0002     MEMADD  RMB     2
```


Within the service routine how will you ensure that the ACIA Transmitter will stop sending characters to the printer, after the last character of the message is printed?

Contd...

Disable the transmitter interrupt by clearing bits #6 and 5 of the ACIA Control Register (see Appendix E). If another device is still operating under interrupt, the above operation will affect only the ACIA transmitter. If the ACIA transmitter was the only interrupting interface, then all IRQ interfaces could be interrupt disabled by the instruction SEI (SEt Interrupt), the opposite to CLI.

11-7

Now write the service routine, entered each time to print one character of the message. Assume the background initialization shown in the previous frame.

```
                                          MPU CYCLES
012A FE 0128 MESPRT LDX     MEMADD    5
012D 08             INX               4
012E FF 0128        STX     MEMADD    6 GET CHAR ADDRESS
0131 A6 00          LDA A   X         5 GET CHAR
0133 27 04          BEQ     NOMORE    4
0135 B7 7FF5        STA A   SERBUF    5 PRINT IT
0138 3B       PRTRTI RTI              10  TOTAL 39 MPU CYCLES
0139 B6 738E NOMORE LDA A   ACIACR
013C 84 9F          AND A   #%10011111
013E B7 738E        STA A   ACIACR    DISABLE TX INT
0141 B7 7FF4        STA A   SERCSR
0144 20 F2          BRA     PRTRTI
```

At slow terminal rates e.g. 10 characters/sec one character is printed every 100 msec. At higher data rates e.g. 960 char/sec, one character is printed every millisecond. The above service routine requires 39 MPU cycles plus 9 to push and interrupt. Assuming approximately 50 MPU cycles per interrupt, this is still only 50 microseconds, using a 1MHz MPU clock. Hence 10 000 to 20 000 interrupts per second are theoretically possible, supporting dozens of devices. Therein lies the power of interrupt.

So far we have looked at only one device operating under interrupt at one time. Consider an ACIA connected to a printer (output) and a keyboard (input), both operating under interrupt. When an IRQ operation is demanded by one of these devices, the first task of the service routine is to determine which device produced the interrupt. This is done by consecutively checking the READY bit of each device capable of IRQ operation.

Write the first part of the IRQ service routine IRQSER which determines whether the ACIA's receiver or transmitter requires service, branching to KEYSER to service the keyboard or PRTSER to service the transmitter.

```
                *
0200 B6 7FF4 IRQSER LDA A  SERCSR
0203 85 01          BIT A  #$01      RX READY?
0205 26 49          BNE    KEYSER
0207 85 02          BIT A  #$02      TX READY?
0209 26 65          BNE    PRTSER
020B 3B      INTRTN RTI               RETURN POINT FOR ALL
```

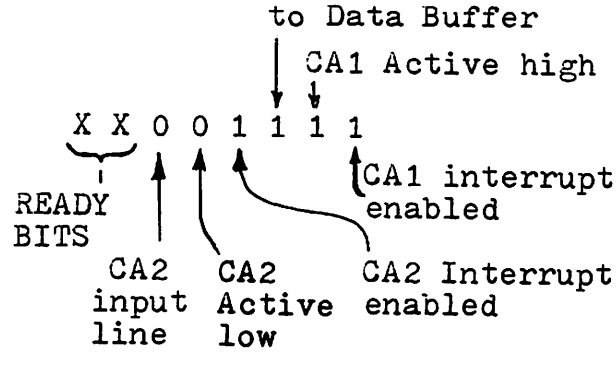
Both service routines would branch back to here.

Although all IRQ controlled devices are theoretically equal for interrupt service it is normal to poll the READY bit of the fastest device first, if one is significantly faster than the other to avoid losing data from the faster device while servicing a slower device. Hence the first device polled effectively has a slightly higher priority, this advantage increasing as more devices requiring IRQ service are added to the system.

PIA Control Lines acting as inputs can produce IRQ operation if enabled for interrupt via the PIA's Control Register. When bit #0 of Control Register A (or B) is set, interrupt is then possible via CA1 (CB1). Similarly CA2 (CB2) is enabled via bit #3. CA2 (CB2) as an output line does not produce an interrupt since interrupts originate with the external device such as a keyboard, telling the computer that data is ready to be moved or that some control action is needed.

Write the background initialization to permit CA1 of the PIA to interrupt when going high (1) and CA2 as an input to interrupt when going low (0). The A half of the PIA should be set to receive 8 bit parallel data.

```
* PROG TO SET UP PIA A HALF AS INPUT
*
PINPUT LDA A  PIACRA
       AND A  #%11001011
       STA A  PIACRA    CLEAR BIT 2
       CLR    PIABPA    INPUT MODE
       ORA A  #%00001111
       STA A  PIACRA
       LDX    #PIASER
       STX    $FFF8
       CLI
HR     BRA    HR
```



When an interrupt is produced by CA1 of the PIA the service routine is to store bits #0 to 3 of the Data Buffer in LODATA. An interrupt by CA2 should store bits #4 to 7 in HIDATA. Assume that CA1 and CA2 are the only source of interrupts. Write the service routines.

```
     0350     HIDATA EQU    $0350
     0352     LODATA EQU    $0352
              *
0100 B6 7FF1 PIASER LDA A  PIACRA
0103 2B 05          BMI    CA1INT   CA1 INT REQUEST VIA BIT 7
0105 85 40          BIT A  #%01000000
0107 26 0B          BNE    CA2INT   CA2 INT REQUEST VIA BIT 6
0109 3B      PIARTN RTI
010A B6 7FF0 CA1INT LDA A  PIABFA
010D 84 0F          AND A  #$0F     ZAP HI BITS
010F B7 0352        STA A  LODATA
0112 20 F5          BRA    PIARTN
0114 B6 7FF0 CA2INT LDA A  PIABFA
0117 84 F0          AND A  #$F0     ZAP LO BITS
0119 B7 0350        STA A  HIDATA
011C 20 EB          BRA    PIARTN
```

If several PIA's are connected as IRQ devices, but capable of interrupt via CA1 only, the skip chain becomes:

```
LDA A  PIACR5
BMI    PIA5
LDA A  PIACR6
BMI    PIA6
    etc.
```

Another major use of IRQ operation is in controlling the timing of specific computer operations. For example a digital voltmeter may be required to make a measurement in a lab experiment or in a process-control operation at the rate of 10 measurements per second. Aside from the inaccuracy of using timing loops for control of these measurements, the computer is not available for other tasks.

The solution is in the use of a "Real Time Clock", a device which produces interrupts at specific times or rates. The service routine for the real time clock would then determine which devices get service at what times. In the example above, the real time clock could be driven by the 60Hz line signal producing 60 interrupts/sec. Write the background initialization and service routine for this clock which causes the digital voltmeter to make 10 measurements per second via the subroutine DVMSER.

```
0100 86 06          LDA A  #$06
0102 B7 011C        STA A  COUNT
0105 CE 010E        LDX    #CLKSER
0108 FF FFF8        STX    $FFF8
010B 0E             CLI
010C 20 FE   HERE   BRA    HERE      SPIN IN BACK
             *
010E 7A 011C CLKSER DEC    COUNT
0111 26 08          BNE    CLKRTN    NOT THIS TIME
0113 86 06          LDA A  #$06      YES. RESET COUNTER
0115 B7 011C        STA A  COUNT
0118 BD 0240        JSR    DVMSER    AND MEASURE VOLTAGE
011B 3B      CLKRTN RTI               ALL DONE
             *
011C 0001    COUNT  RMB    1
```

This line frequency-controlled clock is a very simple timer. Real Time Clocks, much more complex than this, are commercially available.

The Non Maskable Interrupt (NMI) is essentially the same as the IRQ with the following exceptions:

(a) It is always enabled (capable of interrupting), independent of the I bit status.

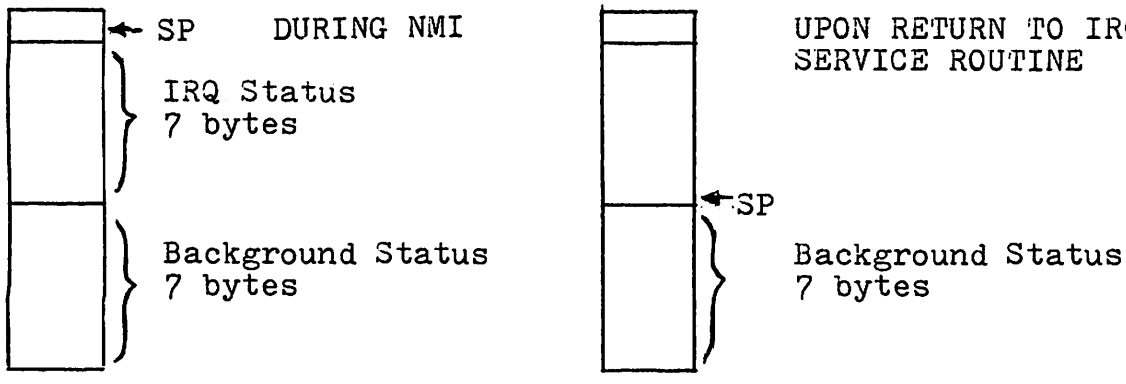
(b) Its vector addresses are FFFC and FFFD.

(c) It will interrupt only when the MPU's $\overline{NMI}$ line changes state from 1 to 0. It will not re-interrupt until after $\overline{NMI}$ has gone high and then is grounded again.

NMI operation is needed when a high speed device requires high priority service, even if an IRQ service routine is presently being executed, in which case the IRQ service routine is interrupted to provide NMI service.

During an NMI service routine all other interrupts are automatically disabled, hence NMI service routines cannot be interrupted even for another NMI device. Upon return from an NMI service routine, service will be provided for another NMI device, if one is waiting; otherwise it will resume service to an interrupted IRQ service routine, if one was interrupted. If none of these are waiting,service will be provided to other waiting IRQ devices, or to a background program, in that order.

Assuming that an NMI device interrupted an IRQ service routine, show the state of the stack (in general terms) during the NMI service routine.



In de-bugging a faulty program it is sometimes necessary to know the status of internal registers (A, B, X, etc.) after execution of a specific instruction within a program. This is possible via the instruction

SWI (SoftWare Interrupt - operation code 3F)

If 3F (SWI) is placed in memory, in the byte following a specific instruction, normal program execution will take place until this 3F is encountered, whereupon all internal registers will be stored on the stack, as if entering an IRQ or NMI service routine. In this case the program will transfer control via vector addresses FFFA and FFFB to the SWI service routine, which usually prints out the contents of the internal registers from the stack. Insertion of the 3F code destroys the original program, hence most systems require RESET after an SWI service routine is executed. An exception to this exists in some de-bugging programs which save the byte which was replaced by 3F, and then restore it after execution of the SWI service routine.

In some 6800 systems where the SWI routine is provided in permanent or "Read Only Memory" (ROM) the vectors for SWI may also be in ROM, rather than in Read/Write Memory, usually called RAM (Random Access Memory), which can be initialized via RESET. If vectors are permanent a user-written SWI routine cannot be implemented.

Why is the stack essential to SWI operation?

Data must be saved by MPU hardware rather than via software (program) which itself would use some of these registers and therefore modify their contents.

Write the background initialization and the SWI service routine to print the contents of CCR, ACC B, and ACC A simply as 6 ASCII characters, one after the other, when SWI is encountered within the program. Assume an available subroutine, OBYTE, which prints 2 ASCII characters, based on the 8 bit contents of ACC A.

```
                    *
                    * SOFTWARE INTERRUPT SERVICE TO PRINT CCR,
                    * ACC A AND B ON CONSOLE TERMINAL.  CALLS OBYTE SUB.
                    *
      FFFA          SWIVEC EQU    $FFFA
                    *
                    * BACKGROUND INITIALIZATION FOR SWI.
0200 CE 0240              LDX    #SWISER
0203 FF FFFA              STX    SWIVEC
0206 20 FE    HR          BRA    HR
                    *
                    * SWI SERVICE ROUTINE
                    *
0240                      ORG    $0240
0240 32       SWISER      PUL A             GET CCR FROM STACK
0241 BD 0142              JSR    OBYTE      PRINT CCR
0244 32                   PUL A
0245 BD 0142              JSR    OBYTE      PRINT B
0248 32                   PUL A
0249 BD 0142              JSR    OBYTE      PRINT A
024C 20 FE    HERE        BRA    HERE
                    *
      0142      OBYTE     EQU    $0142
                          END
```

Now write the first part of a different SWI service routine SOFINT, which prints a more readable output of the stored data, e.g.,

CCR= XX (where XX = stored CCR value)

Assume the following available subroutines:

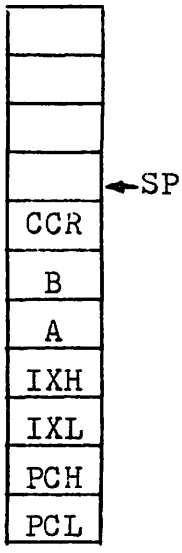
OBYTE - prints contents of ACC A as 2 ASCII character.
OUTMES - prints ASCII message terminated by null. X = pointer.
CRLF - Carriage Return and Line Feed.

```
                    *
                    * PRINTOUT OF REGISTERS AFTER SOFTWARE INTERRUPT
                    *
0200 CE 0250            LDX     #SOFINT
0203 FF FFFA            STX     $FFFA     INIT SWI VECTOR.
                    * NOW JUMP TO TARGET PROGRAM
                    *
0250                    ORG     $0250
0250 BD 0179 SOFINT     JSR     CRLF
0253 CE 0281            LDX     #CCRMES
0256 BD 1F0C            JSR     OUTMES    PRINT CCR=
0259 32                 PUL A
025A BD 0142            JSR     OBYTE     PRINT CCR CONTENTS
                    *
                    *
0281 43         CCRMES  FCC     /CCR= /
0286 00                 FCB     0
```

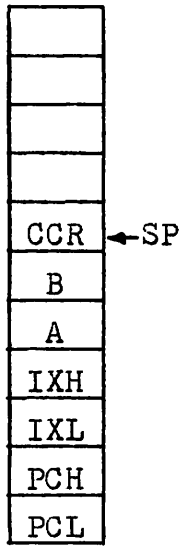
Note that entry to OBYTE is at 0142, rather than at 0139, in the original OBYTE routine (Subroutine Chapter), since the data to be printed is already in ACC A. The CRLF routine is also from the Subroutine Chapter. The OUTMES routine is from the ACIA Chapter, but in subroutine format.

Execution of the OBYTE subroutine involves use of the stack. Will this destroy data now on the stack, yet to be printed within the SWI service routine? Use stack diagrams to prove your answer.

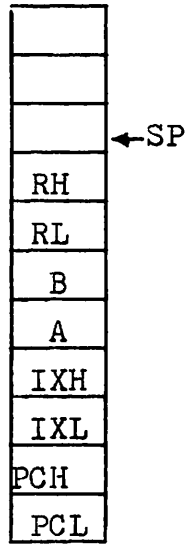
No. Data to be printed will not be destroyed.



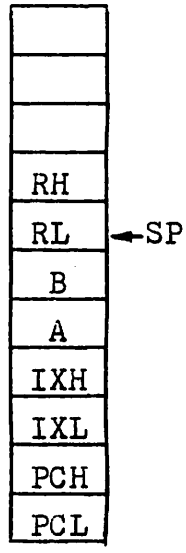
Within SWI service routine before printout begins.



After first PUL A.



Within OBYTE sub. RH and RL are return address bytes. H = high, L = low. CCR data on the stack is overwritten but only after it is in ACC A for printing.



After return from OBYTE subroutine. RL and RH will be overwritten in future use of the stack.

Continuing with the same service routine, assume that CCR, ACC B and ACC A have been pulled and printed on one line. How would you print the Index Register contents, still continuing on the same line? Include the message in your answer.

```
0271 CE 0291          LDX     #IXMESS
0274 BD 1F0C          JSR     OUTMES    PRINT X=
0277 32               PUL A
0278 BD 0142          JSR     OBYTE     PRINT HI BYTE OF X
027B 32               PUL A
027C BD 0142          JSR     OBYTE     PRINT LO BYTE OF X
                *
                *
0291 20       IXMESS  FCC     / X= /
0295 00               FCB     0
```

Note space before and after message to make message readable.

Such a routine is normally included in the computer system software and is essential in "de-bugging" faulty programs. By setting SWI (3F) in the main program, just after a subroutine call, the results of the subroutine can be examined in detail to determine how it performed. The place in the main program where the SWI occurs is often called a breakpoint.

More sophisticated de-bug routines permit multiple breakpoints for testing of partially completed programs, e.g., subroutine calls for which the subroutines have not yet been written. The "loose ends" or unwritten code can be caught by breakpoints.

The complete listing for the SWI de-bug routine is shown below.

```
                        *
     1F0C               OUTMES  EQU     $1F0C
     0179               CRLF    EQU     $0179
     0142               OBYTE   EQU     $0142     LATE ENTRY. AVOIDS X.
                        *
                        * PRINTOUT OF REGISTERS AFTER SOFTWARE INTERRUPT
                        *
0200 CE 0250                    LDX     #SOFINT
0203 FF FFFA                    STX     $FFFA     INIT SWI VECTOR.
                        * NOW JUMP TO TARGET PROGRAM
                        *
0250                            ORG     $0250
0250 BD 0179 SOFINT             JSR     CRLF
0253 CE 0281                    LDX     #CCRMESS
0256 BD 1F0C                    JSR     OUTMES    PRINT CCR=
0259 32                         PUL A
025A BD 0142                    JSR     OBYTE     PRINT CCR CONTENTS
025D CE 028C                    LDX     #BMESS
0260 BD 1F0C                    JSR     OUTMES    PRINT B=
0263 32                         PUL A
0264 BD 0142                    JSR     OBYTE     PRINT B CONTENTS
0267 CE 0287                    LDX     #AMESS
026A BD 1F0C                    JSR     OUTMES    PRINT A=
026D 32                         PUL A
026E BD 0142                    JSR     OBYTE     PRINT A CONTENTS
0271 CE 0291                    LDX     #IXMESS
0274 BD 1F0C                    JSR     OUTMES    PRINT X=
0277 32                         PUL A
0278 BD 0142                    JSR     OBYTE     PRINT HI BYTE OF X
027B 32                         PUL A
027C BD 0142                    JSR     OBYTE     PRINT LO BYTE OF X
027F 20 FE         HERE         BRA     HERE
                        *
0281 43            CCRMES       FCC     /CCR= /
0286 00                         FCB     0
0287 20            AMESS        FCC     / A= /
028B 00                         FCB     0
028C 20            BMESS        FCC     / B= /
0290 00                         FCB     0
0291 20            IXMESS       FCC     / X= /
0295 00                         FCB     0
                                END
```

The final printout of this could then look like:

CCR= 2F B= D3 A= F2 X= 1C55

Congratulations! You have completed the workbook. Good luck with your programs.

RWS

APPENDIX A

Hex Codes - 4 bits

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

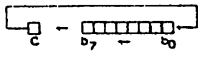
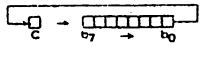
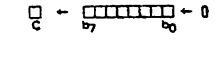
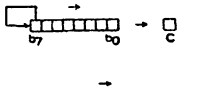
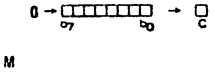
APPENDIX B

ASCII Codes

BITS 4 thru 6 → / BITS 0 thru 3 ↓	—	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P		p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	/	l	/
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	⟨	n	≈
F	SI	US	/	?	O	—	o	DEL

Courtesy Motorola Semiconductor Products, Inc.

Instruction Set (2 pages)

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	5 H	4 I	3 N	2 Z	1 V	0 C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	•	•	↕	↕	R	•
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	•	•	↕	↕	R	•
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	•	•	↕	↕	R	•
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	•	•	↕	↕	R	•
Clear	CLR							6F	7	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	•	•	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	•	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\overline{M} \rightarrow M$	•	•	↕	↕	R	S
	COMA													43	2	1	$\overline{A} \rightarrow A$	•	•	↕	↕	R	S
	COMB													53	2	1	$\overline{B} \rightarrow B$	•	•	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 − M → M	•	•	↕	↕	①	②
	NEGA													40	2	1	00 − A → A	•	•	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	•	•	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	•	•	↕	↕	④	•
	DECA													4A	2	1	A − 1 → A	•	•	↕	↕	④	•
	DECB													5A	2	1	B − 1 → B	•	•	↕	↕	④	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	•	•	↕	↕	R	•
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	•	•	↕	↕	R	•
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	•	•	↕	↕	⑤	•
	INCA													4C	2	1	A + 1 → A	•	•	↕	↕	⑤	•
	INCB													5C	2	1	B + 1 → B	•	•	↕	↕	⑤	•
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	•	•	↕	↕	R	•
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	•	•	↕	↕	R	•
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	•	•	↕	↕	R	•
Push Data	PSHA													36	4	1	$A \rightarrow M_{SP}$, SP − 1 → SP	•	•	•	•	•	•
	PSHB													37	4	1	$B \rightarrow M_{SP}$, SP − 1 → SP	•	•	•	•	•	•
Pull Data	PULA													32	4	1	SP + 1 → SP, $M_{SP} \rightarrow A$	•	•	•	•	•	•
	PULB													33	4	1	SP + 1 → SP, $M_{SP} \rightarrow B$	•	•	•	•	•	•
Rotate Left	ROL							69	7	2	79	6	3				M	•	•	↕	↕	⑥	↕
	ROLA													49	2	1	A 	•	•	↕	↕	⑥	↕
	ROLB													59	2	1	B	•	•	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M	•	•	↕	↕	⑤	↕
	RORA													46	2	1	A 	•	•	↕	↕	⑤	↕
	RORB													56	2	1	B	•	•	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	•	•	↕	↕	⑥	↕
	ASLA													48	2	1	A 	•	•	↕	↕	⑥	↕
	ASLB													58	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	•	•	↕	↕	⑥	↕
	ASRA													47	2	1	A 	•	•	↕	↕	⑥	↕
	ASRB													57	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Logic.	LSR							64	7	2	74	6	3				M	•	•	R	↕	⑥	↕
	LSRA													44	2	1	A 	•	•	R	↕	⑥	↕
	LSRB													54	2	1	B	•	•	R	↕	⑥	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	•	•	↕	↕	R	•
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	•	•	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	•	•	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	•	•	↕	↕	R	•
	TBA													17	2	1	B → A	•	•	↕	↕	R	•
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	•	•	↕	↕	R	R
	TSTA													4D	2	1	A − 00	•	•	↕	↕	R	R
	TSTB													5D	2	1	B − 00	•	•	↕	↕	R	R

Instruction Set (2 pages)

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M+1)$	•	•	⑦	‡	⑧	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	‡	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	‡	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M+1) \rightarrow X_L$	•	•	⑨	‡	R	•
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	•	•	⑨	‡	R	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M+1)$	•	•	⑨	‡	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	•	•	⑨	‡	R	•
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE OP	RELATIVE ~	RELATIVE #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BRANCH TEST	5 H	4 I	3 N	2 Z	1 V	0 C
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•
Branch If ≥ Zero	BGE	2C	4	2										N ⊖ V = 0	•	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										Z + (N ⊖ V) = 0	•	•	•	•	•	•
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•	•
Branch If ≤ Zero	BLE	2F	4	2										Z + (N ⊖ V) = 1	•	•	•	•	•	•
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										N ⊖ V = 1	•	•	•	•	•	•
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				} See Special Operations	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI										38	10	1		—	—	—	—	⑩	—
Return From Subroutine	RTS										39	5	1	} See special Operations	•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1		•	S	•	•	•	•
Wait for Interrupt	WAI										3E	9	1		•	⑪	•	•	•	•

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER OP	INHER ~	INHER =	BOOLEAN OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	•	•	•	•	S	•
Acmltr A → CCR	TAP	06	2	1	$A \rightarrow CCR$	—	—	—	⑫	—	—
CCR → Acmltr A	TPA	07	2	1	$CCR \rightarrow A$	•	•	•	•	•	•

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?
② (Bit C) Test: Result = 00000000?
③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
④ (Bit V) Test: Operand = 10000000 prior to execution?
⑤ (Bit V) Test: Operand = 01111111 prior to execution?
⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred.
⑦ (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?
⑧ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?
⑨ (Bit N) Test: Result less than zero? (Bit 15 = 1)
⑩ (All) Load Condition Code Register from Stack. (See Special Operations)
⑪ (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
⑫ (ALL) Set according to the contents of Accumulator A.

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
#	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
−	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
		R	Reset Always
		S	Set Always
+	Boolean Inclusive OR;	‡	Test and set if true, cleared otherwise
⊕	Boolean Exclusive OR;	•	Not Affected
$\overline{M}$	Complement of M;	CCR	Condition Code Register
→	Transfer Into;	LS	Least Significant
0	Bit = Zero;	MS	Most Significant

Machine Code

00	*		40	NEG	A		80	SUB	A	IMM	C0	SUB	B	IMM
01	NOP		41	*			81	CMP	A	IMM	C1	CMP	B	IMM
02	*		42	*			82	SBC	A	IMM	C2	SBC	B	IMM
03	*		43	COM	A		883	*			C3	*		
04	*		44	LSR	A		84	AND	A	IMM	C4	AND	B	IMM
05	*		45	*			85	BIT	A	IMM	C5	BIT	B	IMM
06	TAP		46	ROR	A		86	LDA	A	IMM	C6	LDA	B	IMM
07	TPA		47	ASR	A		88	*			C7	*		
08	INX		48	ASL	A		88	EOR	A	IMM	C8	EOR	B	IMM
09	DEX		49	ROL	A		89	ADC	A	IMM	C9	ADC	B	IMM
0A	CLV		4A	DEC	A		8A	ORA	A	IMM	CA	ORA	B	IMM
0B	SEV		4B	*			8B	ADD	A	IMM	CB	ADD	B	IMM
0C	CLC		4C	INC	A		8C	CPX	A	IMM	CC	*		
OD	SEC		4D	TST	A		8D	BSR		REL	CD	*		
0E	CLI		4E	*			8E	LDS		IMM	CE	LDX		IMM
0F	SEI		4F	CLR	A		8F	*			CF	*		
10	SBA		50	NEG	B		90	SUB	A	DIR	DO	SUB	B	DIR
11	CBA		52	*			91	CMP	A	DIR	D1	CMP	B	DIR
12	*		52	*			92	SBC	A	DIR	D2	SBC	B	DIR
13	*		53	COM	B		93	*			D3	*		
14	*		54	LSR	B		94	AND	A	DIR	D4	AND	B	DIR
15	*		55	*			95	BIT	A	DIR	D5	BIT	B	DIR
16	TAB		56	ROR	B		96	LDA	A	DIR	D6	LDA	B	DIR
17	TBA		57	ASR	B		97	STA	A	DIR	D7	STA	B	DIR
18	*		58	ASL	B		98	EOR	A	DIR	D8	EOR	B	DIR
19	DAA		59	ROL	B		99	ADC	A	DIR	D9	ADC	B	DIR
1A	*		5A	DEC	B		9A	ORA	A	DIR	DA	ORA	B	DIR
1B	ABA		5B	*			9B	ADD	A	DIR	DB	ADD	B	DIR
1C	*		5C	INC	B		9C	CPX		DIR	DC	*		
1D	*		5D	TST	B		9D	*			DD	*		
1E	*		5E	*			9E	LDS		DIR	DE	LDX		DIR
1F	*		5F	CLR	B		9F	STS		DIR	DF	STX		DIR
20	BRA	REL	60	NEG	IND		A0	SUB	A	IND	E0	SUB	B	IND
21	*		61	*			A1	CMP	A	IND	E1	CMP	B	IND
22	BHI	REL	62	*			A2	SBC	A	IND	E2	SBC	B	IND
23	BLS	REL	63	COM	IND		A3	*			E3	*		
24	BCC	REL	64	LSR	IND		A4	AND	A	IND	E4	AND	B	IND
25	BCS	REL	65	*			A5	BIT	A	IND	E5	BIT	B	IND
26	BNE	REL	66	ROR	IND		A6	LDA	A	IND	E6	LDA	B	IND
27	BEQ	REL	67	ASR	IND		A7	STA	A	IND	E7	STA	B	IND
28	BVC	REL	68	ASL	IND		A8	EOR	A	IND	E8	EOR	B	IND
29	BVS	REL	69	ROL	IND		A9	ADC	A	IND	E9	ADC	B	IND
2A	BPL	REL	6A	DEC	IND		AA	ORA	A	IND	EA	ORA	B	IND
2B	BMI	REL	6B	*			AB	ADD	A	IND	EB	ADD	B	IND
2C	BGE	REL	6C	INC	IND		AC	CPX		IND	EC	*		
2D	BLT	REL	6D	TST	IND		AD	JSR		IND	ED	*		
2E	BGT	REL	6E	JMP	IND		AE	LDS		IND	EE	LDX		IND
2F	BLE	REL	6F	CLR	IND		AF	STS		IND	EF	STX		IND
30	TSX		70	NEG	EXT		B0	SUB	A	EXT	F0	SUB	B	EXT
31	INS		71	*			B1	CMP	A	EXT	F1	CMP	B	EXT
32	PUL	A	72	*			B2	SBC	A	EXT	F2	SBC	B	EXT
33	PUL	B	73	COM	EXT		B3	*			F3	*		
34	DES		74	LSR	EXT		B4	AND	A	EXT	F4	AND	B	EXT
35	TXS		75	*			B5	BIT	A	EXT	F5	BIT	B	EXT
36	PSH	A	76	ROR	EXT		B6	LDA	A	EXT	F6	LDA	B	EXT
37	PSH	B	77	ASR	EXT		B7	STA	A	EXT	F7	STA	B	EXT
38	*		78	ASL	EXT		B8	EOR	A	EXT	F8	ADC	B	EXT
39	RTS		79	ROL	EXT		B9	ADC	A	EXT	F9	ADC	B	EXT
3A	*		7A	DEC	EXT		BA	ORA	A	EXT	FA	ORA	B	EXT
3B	RTI		7B	*			BB	ADD	A	EXT	FB	ADD	B	EXT
3C	*		7C	INC	EXT		BC	CPX		EXT	FC	*		
3D	*		7D	TST	EXT		BD	JSR		EXT	FD	*		
3E	WAI		7E	JMP	EXT		BE	LDS		EXT	FE	LDX		EXT
3F	SWI		7F	CLR	EXT		BF	STS		EXT	FF	STX		EXT

Notes: 1. Addressing Modes: A = Accumulator A; B = Accumulator B; IMM = Immediate; DIR = Direct; EXT = Extended; REL = Relative; IND = Indexed

2. Unassigned code indicated by "*"

Hexadecimal Values of Machine Codes

- ACIA -

Asynchronous Communications Interface Adapter

DEFINITION OF ACIA REGISTER CONTENTS

Data Bus Line Number	Buffer Address			
	Transmit Data Register	Receive Data Register	Control Register	Status Register
	(Write Only)	(Read Only)	(Write Only)	(Read Only)
0	Data Bit 0*	Data Bit 0	Counter Divide Select 1 (CR0)	Receive Data Register Full (RDRF)
1	Data Bit 1	Data Bit 1	Counter Divide Select 2 (CR1)	Transmit Data Register Empty (TDRE)
2	Data Bit 2	Data Bit 2	Word Select 1 (CR2)	$\overline{\text{Data Carrier Detect}}$ ($\overline{\text{DCD}}$)
3	Data Bit 3	Data Bit 3	Word Select 2 (CR3)	$\overline{\text{Clear-to-Send}}$ ($\overline{\text{CTS}}$)
4	Data Bit 4	Data Bit 4	Word Select 3 (CR4)	Framing Error (FE)
5	Data Bit 5	Data Bit 5	Transmit Control 1 (CR5)	Receiver Overrun (OVRN)
6	Data Bit 6	Data Bit 6	Transmit Control 2 (CR6)	Parity Error (PE)
7	Data Bit 7***	Data Bit 7**	Receive Interrupt Enable (CR7)	Interrupt Request (IRQ)

* Leading bit = LSB = Bit 0
** Data bit will be zero in 7-bit plus parity modes.
*** Data bit is "don't care" in 7-bit plus parity modes.

ACIA Control Register Format

Enable for Receiver Interrupt

b7 = 1: Enables Interrupt Output in Receiving Mode

b7 = 0: Disables Interrupt Output in Receiving Mode

Counter ratio and Master reset select used in both transmitters and receiver sections

b1	b0	Function (Tx, Rx)
0	0	÷1
0	1	÷16
1	0	÷64
1	1	MASTER RESET

b7	b6	b5	b4	b3	b2	b1	b0
RIE	TC2	TC1	WS3	WS2	WS1	CDS2	CDS1

Transmitter Control Bits: Controls the Interrupt Output* and RTS Output, and provides for Transmission of a Break

b6	b5	Function
0	0	Sets RTS = 0 and inhibits Tx interrupt (TIE)
0	1	Sets RTS = 0 and enables Tx interrupt (TIE)
1	0	Sets RTS = 1 and inhibits Tx interrupt (TIE)
1	1	Sets RTS = 0, Transmits Break and inhibits Tx interrupt (TIE)

*TIE is the enable for the interrupt output in transmit mode.

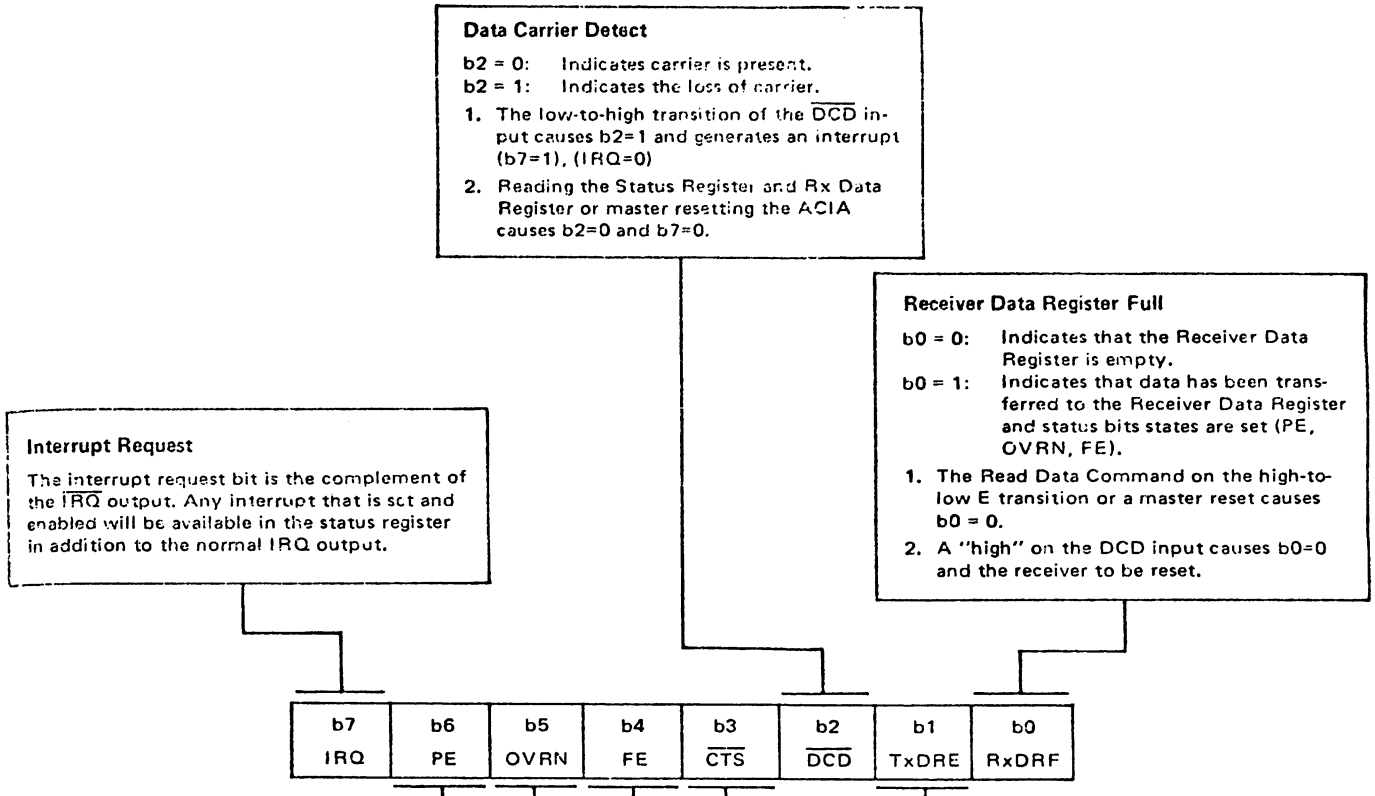
Word Length, Parity, and Stop Bit Select

b4	b3	b2	Word Length	Parity	Stop Bits
0	0	0	7	Even	2
0	0	1	7	Odd	2
0	1	0	7	Even	1
0	1	1	7	Odd	1
1	0	0	8	None	2
1	0	1	8	None	1
1	1	0	8	Even	1
1	1	1	8	Odd	1

- ACIA -

Asynchronous Communications Interface Adapter

ACIA Status Register Format



Data Carrier Detect

b2 = 0: Indicates carrier is present.
b2 = 1: Indicates the loss of carrier.

1. The low-to-high transition of the $\overline{DCD}$ input causes b2=1 and generates an interrupt (b7=1), (IRQ=0)
2. Reading the Status Register and Rx Data Register or master resetting the ACIA causes b2=0 and b7=0.

Receiver Data Register Full

b0 = 0: Indicates that the Receiver Data Register is empty.
b0 = 1: Indicates that data has been transferred to the Receiver Data Register and status bits states are set (PE, OVRN, FE).

1. The Read Data Command on the high-to-low E transition or a master reset causes b0 = 0.
2. A "high" on the DCD input causes b0=0 and the receiver to be reset.

Interrupt Request

The interrupt request bit is the complement of the $\overline{IRQ}$ output. Any interrupt that is set and enabled will be available in the status register in addition to the normal IRQ output.

Framing Error

b4 = 1: Indicates the absence of the first stop bit resulting from character synchronization error, faulty transmission, or a Break condition.

1. The internal Rx data transfer signal causes b4=1 due to the above conditions and causes b4=0 on the next Rx data transfer signal if conditions have been rectified.

Overrun Error

b5 = 1: Indicates that a character or a number of characters were received but not read from the Rx data register prior to subsequent characters being received.

1. The Read Data Command on the high-to-low E transition causes b5=1 and b0=1 if an overrun condition exists. The next Read Data Command on the high-to-low E transition causes b5=0 and b0=0.

Parity Error

b6 = 1: Indicates that a parity error exists. The parity error bit is inhibited if no parity is selected.

1. The parity error status is updated during the internal receiver data transfer signal.

Transmitter Data Register Empty

b1 = 1: Indicates that the transmitter data Register is empty.
b1 = 0: Indicates that the transmitter data Register is full.

1. The internal Tx transfer signal forces b1=1.
2. The Write Data Command on the high-to-low E transition causes b1=0.
3. A "high" on the $\overline{CTS}$ input causes b1=0.

Clear to Send

The $\overline{CTS}$ bit reflects the $\overline{CTS}$ input status for use by the MPU for interfacing to a modem.

NOTE: The $\overline{CTS}$ input does not reset the transmitter.

- PIA -

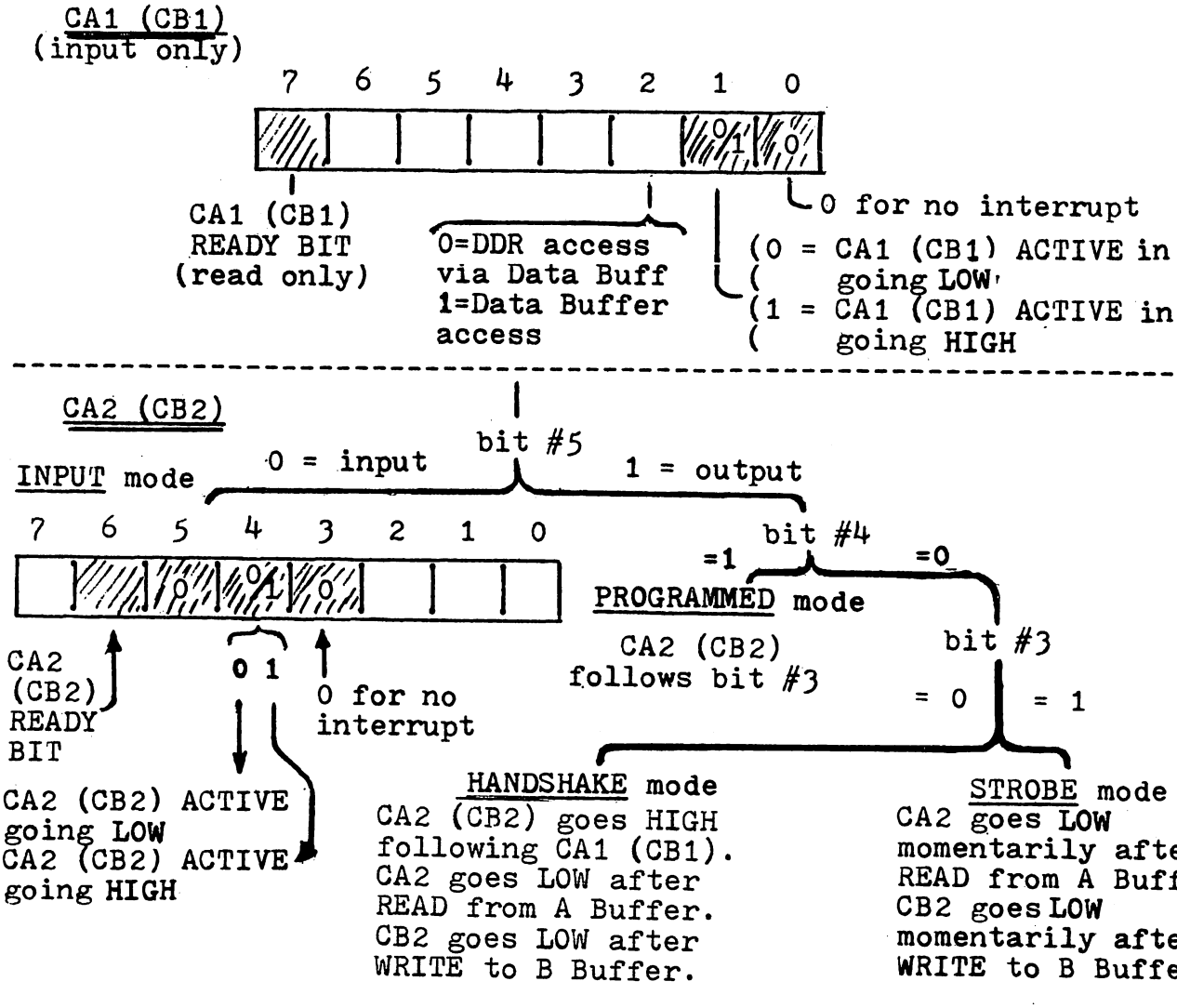
Peripheral Interface Adapter

DATA DIRECTION REGISTER

Accessed via Data Buffer address when bit #2 of the Control Register is 0.

1 = output
0 = input
} for each of the 8 data lines on the Data Buffer.

CONTROL REGISTER



- PIA -
Peripheral Interface Adapter

Determine Active CA1 (CB1) Transition for Setting Interrupt Flag IRQA(B)1 – (bit b7)

b1 = 0 : IRQA(B)1 set by high-to-low transition on CA1 (CB1).

b1 = 1 : IRQA(B)1 set by low-to-high transition on CA1 (CB1).

CA1 (CB1) Interrupt Request Enable/Disable

b0 = 0 : Disables IRQA(B) MPU Interrupt by CA1 (CB1) active transition.[1]

b0 = 1 : Enable IRQA(B) MPU Interrupt by CA1 (CB1) active transition.

1. IRQA(B) will occur on next (MPU generated) positive transition of b0 if CA1 (CB1) active transition occurred while interrupt was disabled.

IRQA(B) 1 Interrupt Flag (bit b7)

Goes high on active transition of CA1 (CB1); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

b7	b6	b5	b4	b3	b2	b1	bØ
IRQA(B)1 Flag	IRQA(B)2 Flag	CA2(CB2) Control			DDR Access	CA1(CB1) Control	

IRQA(B)2 Interrupt Flag (bit b6)

CA2 (CB2) Established as Input (b5 = 0): Goes high on active transition of CA2 (CB2); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

CA2 (CB2) Established as Output (b5 = 1): IRQA(B)2 = 0, not affected by CA2 (CB2) transitions.

Determines Whether Data Direction Register Or Output Register is Addressed

b2 = 0 : Data Direction Register selected.

b2 = 1 : Output Register selected.

CA2 (CB2) Established as Output by b5 = 1

b5 b4 b3 (Note that operation of CA2 and CB2 output functions are not identical)

1 0

→ CA2

b3 = 0 : Read Strobe With CA1 Restore

CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next active CA1 transition.

b3 = 1 : Read Strobe with E Restore

CA2 goes low on first high-to-low E transition following an MPU Read of Output Register A; returned high by next high-to-low E transition.

→ CB2

b3 = 0 : Write Strobe With CB1 Restore

CB2 goes on low on first low-to high E transition following an MPU Write into Output Register B; returned high by the next active CB1 transition.

b3 = 1 : Write Strobe With E Restore

CB2 goes low on first low-to-high E transition following an MPU Write into Output Register B; returned high by the next low-to-high E transition.

b5 b4 b3

1 1

→ Set/Reset CA2 (CB2)

CA2 (CB2) goes low as MPU writes b3 = 0 into Control Register.

CA2 (CB2) goes high as MPU writes b3 = 1 into Control Register.

CA2 (CB2) Established as Input by b5 = 0

b5 b4 b3

0

→ CA2 (CB2) Interrupt Request Enable/ Disable

b3 = 0 : Disables IRQA(B) MPU Interrupt by CA2 (CB2) active transition.[1]

b3 = 1 : Enables IRQA(B) MPU Interrupt by CA2 (CB2) active transition.

1. IRQA(B) will occur on next (MPU generated) positive transition of b3 if CA2 (CB2) active transition occurred while interrupt was disabled.

→ Determines Active CA2 (CB2) Transition for Setting Interrupt Flag IRQA(B)2 – (bit b6)

b4 = 0 : IRQA(B)2 set by high-to-low transition on CA2 (CB2).

b4 = 1 : IRQA(B)2 set by low-to-high transition on CA2 (CB2).

APPENDIX G

CHARACTER SET

The characters used in the source language for the Motorola assembler form a sub-set of ASCII (American Standard Code for Information Interchange, 1968). The ASCII Code is shown in App B.. The following characters are recognized by the assembler:

1. The alphabet A through Z
2. The integers 0 through 9
3. Four arithmetic operators:
   + - * /
4. Characters used as special prefixes:
   - # (pounds sign) specifies the immediate mode of addressing
   - $ (dollar sign) specifies a hexadecimal number
   - @ (commercial at) specifies an octal number
   - % (percent) specifies a binary number
   - ' (apostrophe) specifies an ASCII literal character
5. Characters used as special suffices:
   - B (letter B) specifies a binary number
   - H (letter H) specifies a hexadecimal number
   - O (letter O) specifies an octal number
   - Q (letter Q) specifies a octal number
6. Four separating characters:
   - SPACE
   - Horizontal TAB
   - CR (carriage return)
   - , (comma)

   The use of horizontal TAB is always optional, and can be replaced by SPACE.

Courtesy Motorola Semiconductor Products, Inc.

APPENDIX H

Commonly Used Instructions

As a quick reference guide some of the more commonly used instructions, along with their machine codes, are shown here.

```
                *
86 4C           *       LDA A   #$4C
                *
B7 12F3         *       STA A   $12F3
                *
B6 12F3         *       LDA A   $12F3
                *
FE 12A7         *       LDX     $12A7
                *
08              *       INX
                *
FF 12A7         *       STX     $12A7
                *
A7 00           *       STA A   X
                *
B7 12D5         *       STA A   $12D5
                *
A6 00           *       LDA A   X
```

MICROPROCESSOR GLOSSARY

ACCUMULATOR: The register where arithmetic or logic results are held. Most MPU instructions manipulate or test the accumulator contents.

ACCESS TIME: Time take for specific byte of storage to become available to processor.

ACIA: Asynchronous Communication Inter-face Adapter. Inter-face between asynchronous peripheral and an MPU.

ALU: Arithmetic and Logic Unit. The part of the MPU where arithmetic and logic functions are performed.

ASCII: American Standard Code for Information Interchange. Binary code to represent alphanumeric, special and control characters.

ASSEMBLER: Software which converts assembly language statements into machine code and checks for non valid statements or incomplete definitions.

ASSEMBLY LANG: Means of representing programme statements in mnemonics and conveniently handling memory addressing by use of symbolic terms.

ASYNCHRONOUS: Operations that initiate a new operation immediately upon completion of current one — not timed by system clock.

BASIC: Beginner's All Purpose Symolic Instruction Code. An easy to learn, widely used high level language.

BAUD: Measure of speed of transmission line. Number of times a line changes state per second. Equal to bits per second if each line state represents logic 0 or 1.

BAUDOT CODE: 5-bit code used to encode alphanumeric data.

BCD: Binary Coded Decimal. Means of representing decimal numbers where each figure is replaced by a binary equivalent.

BENCHMARK: A common task for the implementation of which programmes can be written for different MPUs in order to determine the efficiency of the different MPUs in the particular application.

BINARY: The two base number system. The digits are 0 or 1. They are used inside a computer to represent the two states of an electric circuit.

BIT: A single binary digit.

BREAKPOINT: Program address at which execution will be halted to allow debugging or data entry.

BUFFER: Circuit to provide isolation between sensitive parts of a system and the rest of that system.

BUG: A program error that causes the program to malfunction.

BUS: The interconnections in a system that carry parallel binary data. Several bus users are connected to the bus, but generally only one "sender" and one "receiver" are active at any one instant.

BYTE: A group of bits — the most common byte size is eight bits.

CLOCK: The basic timing for a MPU chip.

COMPILER: Software which converts high level language statements into either assembly language statements, or into machine code.

CPU: Central processor unit. The part of a system which performs calculation and data manipulation functions.

CROM: Control Read Only Memory.

CRT: Cathode Ray Tube. Often taken to mean complete output device.

CUTS: Computer Users Tape System. Definition of system for storing data on cassette tape as series of tones to represent binary 1's and 0's.

DEBUG: The process of checking and correcting any program errors either in writing or in actual function.

DIRECT ADDRESSING: An addressing mode where the address of the operand is contained in the instruction. (Address below 100 in 6800)

DMA: Direct Memory Access.

DUPLEX: Transfer of data in two directions simultaneously.

ENVIRONMENT: The conditions of all registers, flags, etc., at any instant in program.

EPROM: Electrically Programmable Read Only Memory. Memory that may be erased (usually by ultra violet light) and reprogrammed electrically.

EXECUTE: To perform a sequence of program steps.

EXECUTION TIME: The time taken to perform an instruction in terms of clock cycles.

FIRMWARE: Instructions or data permanently stored in ROM.

FLAG: A flip flop that may be set or reset under software control.

FLIP-FLOP: Two state device that changes state when clocked.

FLOPPY (DISK): Mass storage which makes use of flexible disks made of a material similar to magnetic tape.

FLOW CHART: A diagram representing the logic of a computer program.

GLITCH: Noise pulse.

HALF DUPLEX: Data transfer in two directions but only one way at a time.

HAND SHAKE: System of data transfer between CPU and peripheral whereby CPU "asks" peripheral if it will accept data and only transfers data if "answer" is yes.

HARD COPY: System output that is printed on paper.

HARDWARE: All the electronic and mechanical components making up a system.

HARD WIRE: Circuits that are comprised of logic gates wired together, the wiring pattern determining the overall logic operation.

HASH: Noisy signal.

HEXADECIMAL: The base 16 number system. Character set is decimal 0 to 9 and letters A to F.

HIGH LEVEL LANGUAGE: Computer language that is easy to use, but which requires compiling into machine code before it can be used by an MPU.

HIGHWAY: As BUS.

IMMEDIATE ADDRESSING: Addressing mode which uses part of the instruction itself as the operand data.

INDEXED ADDRESSING: A form of indirect addressing which uses an Index Register to hold the address of the operand.

INDIRECT ADDRESSING: Addressing mode where the address of the location where the address of the operand may be found is contained in the instruction.

INITIALISE: Set up all registers, flag, etc., to defined conditions.

INSTRUCTION: Bit pattern which must be supplied to an MPU to cause it to perform a particular function.

INSTRUCTION REGISTER: MPU register which is used to hold instructions fetched from memory.

INSTRUCTION SET: The repertoire of instructions that a given MPU can perform.

INTERFACE: Circuit which connects different parts of system together and performs any processing of signals in order to make transfer possible (ie, serial — parallel conversion).

INTERPRETER: An interpreter is a software routine which accepts and executes a high level language program, but unlike a compiler does not produce intermediate machine code listing but converts each instruction as received.

INTERRUPT: A signal to the MPU which will cause it to change from its present task to another.

I/O: Input/Output.

K: Abbreviation for $2^{10} = 1024$

KANSAS CITY (Format): Definition of a CUTS based cassette interface system.

LANGUAGE: A systemmatic means of communicating with an MPU.

LATCH: Retains previous input state until overwritten.

LIFO: Last In First Out. Used to describe data stack.

LOOPING: Program technique where one section of program (the loop) is performed many times over.

MACHINE LANG: The lowest level of program. The only language an MPU can understand without interpreter.

MASK: Bit pattern used in conjunction with a logic operation to select a particular bit or bits from machine word.

MEMORY: The part of a system which stores data (working data or instruction object code).

MEMORY MAP: Chart showing the memory allocation of a system.

MEMORY MAPPED I/O: A technique of implementing I/O facilities by addressing I/O ports as if they were memory locations.

MICRO CYCLE: Single program step in an MPUs Micro program. The smallest level of machine program step.

MICRO PROCESSOR: A CPU implemented by use of large scale integrated circuits. Frequently implemented on a single chip.

MICRO PROGRAM: Program inside MPU which controls the MPU chip during its basic fetch/execute sequence.

MNEMONIC: A word or phrase which stands for another (longer) phrase and is easier to remember.

MODEM: Modulator/demodulator used to send and receive serial data over an audio link.

NON VOLATIVE: Memory which will retain data content after power supply is removed, e.g. ROM.

OBJECT CODE: To bit patterns that are presented to the MPU as instructions and data.

O/C: Open Collector. Means of tieing together O/P's from different devices on the same bus.

OCTAL: Base 8 number system. Character set is decimal 0-8.

OP CODE: Operation Code. A bit pattern which specifies a machine operation in the CPU.

OPERAND: Data used by machine operations.

PARALLEL: Transfer of two or more bits at the same time.

PARITY: Check bit added to data, can be odd or even parity. In odd parity sum of data 1's + parity bit is odd.

PERIPHERAL: Equipment for inputing to or outputting from the system (e.g., teletype, VDU, etc.).

PIA: Peripheral Interface Adapter.

POP: Operation of removing data word from LIFO stack.

PORT: A terminal which the MPU uses to communicate with the outside world.

PROGRAMS: Set of MPU instructions which instruct the MPU to carry out a particular task.

PROGRAM COUNTER: Register which holds the address of next instruction (or data word) of the program being executed.

PROM: Programmable read only memory. Proms are special form of ROM, which can be individually programmed by user.

PUSH: Operation of putting data to LIFO stack.

RAM: Random Access Memory. Read write memory. Data may be written to or read from any location in this type of memory.

REGISTER: General purpose MPU storage location that will hold one MPU word.

RELATIVE ADDRESSING: Mode of addressing whereby address of operand is formed by combining current program count with a displacement value which is part of the instruction.

ROM: Read Only Memory. Memory device which has its data content established as part of manufacture and cannot be changed.

SCRATCH PAD: Memory that has short access time and is used by system for short term data storage.

SERIAL: Transfer of data one bit at a time.

SIMPLEX: Data transmission in one direction only.

SOFTWARE: Programs stored on any media.

SOURCE CODE: The list of statements that make up a program.

STACK: A last in first out store made up of registers or memory locations used for stack.

STATUS REGISTER: Register that is used to store the condition of the accumulator after an instruction has been performed (e.g., Acc = 0).

SUB ROUTINE: A sequence of instructions which perform an often required function, which can be called from any point in the main program.

SYNTAX: The grammar of a programming language.

TRAP (Vector): Pre-defined location in memory which the processor will read as a result of particular condition or operation.

TRI STATE: Description of logic devices whose outputs may be disabled by placing them in a high impedance state.

TTY: Teletype.

TWO'S COMPLEMENT ARITHMETIC: System of performing signed arithmetic with binary numbers.

UART: Universal Asynchronous Receiver Transmitter.

VDU: Video Display Unit.

VECTOR: Memory address, provided to the processor to direct it to a new area in memory.

VOLATILE: Memory devices that will lose data content if power supply removed (i.e., RAM).

WORD: Parallel collection of binary digits much as byte.

Reprinted from the September 1977 edition of Electronics Today International magazine, Toronto, Ontario.

Assembler Error Codes

201 NAM DIRECTIVE ERROR
MESSAGE: ****ERROR 201 AAAAAA
MEANING: THE NAM DIRECTIVE IS NOT THE FIRST SOURCE STATEMENT, IT IS MISSING, OR IT OCCURS MORE THAN ONCE IN THE SAME SOURCE PROGRAM.

202 LABEL OR OPCODE ERROR
MESSAGE: ****ERROR 202 AAAAAA
MEANING: THE LABEL OR OPCODE SYMBOL DOES NOT BEGIN WITH AN ALPHABETIC CHARACTER.

203 STATEMENT ERROR
MESSAGE: ****ERROR 203 AAAAAA
MEANING: THE STATEMENT IS BLANK OR ONLY CONTAINS A LABEL.

204 SYNTAX ERROR
MESSAGE: ****ERROR 204 AAAAAA
MEANING: THE STATEMENT IS SYNTACTICALLY INCORRECT.

205 LABEL ERROR
MESSAGE: ****ERROR 205 AAAAAA
MEANING: THE STATEMENT LABEL FIELD IS NOT TERMINATED WITH A SPACE.

206 REDEFINED SYMBOL
MESSAGE: ****ERROR 206 AAAAAA
MEANING: THE SYMBOL HAS PREVIOUSLY BEEN DEFINED. THE FIRST VALUE IS SAVED IN SYMBOL TABLE.

207 UNDEFINED OPCODE
MESSAGE: ****ERROR 207 AAAAAA
MEANING: THE SYMBOL IN THE OPCODE FIELD IS NOT A VALID OPCODE MNEMONIC OR DIRECTIVE.

208 BRANCH ERROR
MESSAGE: ****ERROR 208 AAAAAA
MEANING: THE BRANCH COUNT IS BEYOND THE RELATIVE BYTE'S RANGE. THE ALLOWABLE RANGE IS:

(*+2) - 126 < D < (*+2) + 127

WHERE: * = ADDRESS OF THE FIRST BYTE OF THE BRANCH INSTRUCTION
D = ADDRESS OF THE DESTINATION OF THE BRANCH INSTRUCTION.

209 ILLEGAL ADDRESS MODE
MESSAGE: ****ERROR 209 AAAAAA
MEANING: THE MODE OF ADDRESSING IS NOT ALLOWED WITH THE OP-CODE TYPE.

210 BYTE OVERFLOW
MESSAGE: ****ERROR 210 AAAAAA
MEANING: AN EXPRESSION CONVERTED TO A VALUE GREATER THAN 255 (DECIMAL). THIS ERROR ALSO OCCURS ON COMPUTER SYSTEMS HAVING WORD LENGTHS OF 16 BITS WHEN USING NEGATIVE OPERANDS IN THE IMMEDIATE ADDRESSING MODE. EXAMPLE:

```
LDA A   #-5     ; CAUSES ERROR 210
```

THE ERROR MAY BE AVOIDED BY USING THE 8 BIT TWO'S COMPLEMENT OF THE NUMBER. EXAMPLE:

```
LDA A   #$FB    ; ASSEMBLES OK
```

Assembler Error Codes

211 UNDEFINED SYMBOL
MESSAGE: ****ERROR 211 AAAAAA
MEANING: THE SYMBOL DOES NOT APPEAR IN A LABEL FIELD.

212 DIRECTIVE OPERAND ERROR
MESSAGE: ****ERROR 212 AAAAAA
MEANING: SYNTAX ERROR IN THE OPERAND FIELD OF A DIRECTIVE.

213 EQU DIRECTIVE SYNTAX ERROR
MESSAGE: ****ERROR 213 AAAAAA
MEANING: THE STRUCTURE OF THE EQU DIRECTIVE IS SYNTACTICALLY INCORRECT OR IT HAS NO LABEL.

214 FCB DIRECTIVE SYNTAX ERROR
MESSAGE: ****ERROR 214 AAAAAA
MEANING: THE STRUCTURE OF THE FCB DIRECTIVE IS SYNTACTICALLY INCORRECT.

215 FDB DIRECTIVE SYNTAX ERROR
MESSAGE: ****ERROR 215 AAAAAA
MEANING: THE STRUCTURE OF THE FDB DIRECTIVE IS SYNTACTICALLY INCORRECT.

216 DIRECTIVE OPERAND ERROR
MESSAGE: ****ERROR 216 AAAAAA
MEANING: THE DIRECTIVE'S OPERAND FIELD IS IN ERROR.

217 OPT DIRECTIVE ERROR
MESSAGE: ****ERROR 217 AAAAAA
MEANING: THE STRUCTURE OF THE OPT DIRECTIVE IS SYNTACTICALLY INCORRECT OR THE OPTION IS UNDEFINED.

220 PHASING ERROR
MESSAGE: ****ERROR 220 AAAAAA
MEANING: THE VALUE OF THE P COUNTER DURING PASS 1 AND PASS 2 FOR THE SAME INSTRUCTION IS DIFFERENT.

221 SYMBOL TABLE OVERFLOW
MESSAGE: ****ERROR 221 AAAAAA
MEANING: THE SYMBOL TABLE HAS OVERFLOWED. THE NEW SYMBOL WAS NOT STORED AND ALL REFERENCES TO IT WILL BE FLAGGED AS AN ERROR.

222 SYNTAX ERROR IN THE SYMBOL
MESSAGE: ****ERROR 222 AAAAAA
MEANING: THE ONE-CHARACTER SYMBOLS A, B, AND X CANNOT BE USED FOR USER-DEFINED SYMBOLS. THEIR USE IS RESTRICTED FOR REFERENCES TO THE ACCUMULATORS (A & B) AND TO THE INDEX REGISTER (X). ERROR 222 ALSO FLAGS ALL SOURCE STATEMENTS CONTAINING A SYMBOL THAT HAS BEEN REDEFINED.

223 THE DIRECTIVE CANNOT HAVE A LABEL
MESSAGE: ****ERROR 223 AAAAAA
MEANING: THE DIRECTIVE CANNOT HAVE A LABEL. THE LABEL FIELD MUST BE EMPTY (BLANK).

Instruction Set (spare copy)

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	ADDRESSING MODES: IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG. 5 H	4 I	3 N	2 Z	1 V	0 C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	•	•	↕	↕	R	•
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	•	•	↕	↕	R	•
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	•	•	↕	↕	R	•
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	•	•	↕	↕	R	•
Clear	CLR							6F	7	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	•	•	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	•	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\bar{M} \rightarrow M$	•	•	↕	↕	R	S
	COMA													43	2	1	$\bar{A} \rightarrow A$	•	•	↕	↕	R	S
	COMB													53	2	1	$\bar{B} \rightarrow B$	•	•	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 − M → M	•	•	↕	↕	①	②
	NEGA													40	2	1	00 − A → A	•	•	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	•	•	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	•	•	↕	↕	④	•
	DECA													4A	2	1	A − 1 → A	•	•	↕	↕	④	•
	DECB													5A	2	1	B − 1 → B	•	•	↕	↕	④	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	•	•	↕	↕	R	•
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	•	•	↕	↕	R	•
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	•	•	↕	↕	⑤	•
	INCA													4C	2	1	A + 1 → A	•	•	↕	↕	⑤	•
	INCB													5C	2	1	B + 1 → B	•	•	↕	↕	⑤	•
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	•	•	↕	↕	R	•
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	•	•	↕	↕	R	•
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	•	•	↕	↕	R	•
Push Data	PSHA													36	4	1	$A \rightarrow M_{SP}$, SP − 1 → SP	•	•	•	•	•	•
	PSHB													37	4	1	$B \rightarrow M_{SP}$, SP − 1 → SP	•	•	•	•	•	•
Pull Data	PULA													32	4	1	SP + 1 → SP, $M_{SP} \rightarrow A$	•	•	•	•	•	•
	PULB													33	4	1	SP + 1 → SP, $M_{SP} \rightarrow B$	•	•	•	•	•	•
Rotate Left	ROL							69	7	2	79	6	3				M	•	•	↕	↕	⑥	↕
	ROLA													49	2	1	A (C, b_7 … b_0)	•	•	↕	↕	⑥	↕
	ROLB													59	2	1	B	•	•	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M	•	•	↕	↕	⑥	↕
	RORA													46	2	1	A (C, b_7 … b_0)	•	•	↕	↕	⑥	↕
	RORB													56	2	1	B	•	•	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	•	•	↕	↕	⑥	↕
	ASLA													48	2	1	A (C ← b_7 … b_0 ← 0)	•	•	↕	↕	⑥	↕
	ASLB													58	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	•	•	↕	↕	⑥	↕
	ASRA													47	2	1	A (b_7 … b_0 → C)	•	•	↕	↕	⑥	↕
	ASRB													57	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Logic.	LSR							64	7	2	74	6	3				M	•	•	R	↕	⑥	↕
	LSRA													44	2	1	A (0 → b_7 … b_0 → C)	•	•	R	↕	⑥	↕
	LSRB													54	2	1	B	•	•	R	↕	⑥	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	•	•	↕	↕	R	•
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	•	•	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	•	•	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	•	•	↕	↕	R	•
	TBA													17	2	1	B → A	•	•	↕	↕	R	•
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	•	•	↕	↕	R	R
	TSTA													4D	2	1	A − 00	•	•	↕	↕	R	R
	TSTB													5D	2	1	B − 00	•	•	↕	↕	R	R

Instruction Set (spare copy)

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M+1)$	•	•	⑦	↕	⑧	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	↕	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	↕	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M+1) \rightarrow X_L$	•	•	⑨	↕	R	•
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	•	•	⑨	↕	R	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M+1)$	•	•	③	↕	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	•	•	③	↕	R	•
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE OP	RELATIVE ~	RELATIVE #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BRANCH TEST	5 H	4 I	3 N	2 Z	1 V	0 C
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	•	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	•	•	•	•	•	•
Branch If Higher — unsigned	BHI	22	4	2										$C + Z = 0$	•	•	•	•	•	•
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	•	•	•	•	•	•
Branch If Lower Or Same — unsigned	BLS	23	4	2										$C + Z = 1$	•	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	•	•	•	•	•	•
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI										3B	10	1		—	—	—	—	⑩	—
Return From Subroutine	RTS										39	5	1	See special Operations	•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1		•	S	•	•	•	•
Wait for Interrupt	WAI										3E	9	1		•	⑪	•	•	•	•

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER OP	INHER ~	INHER =	BOOLEAN OPERATION	5 H	4 I	3 N	2 Z	1 V	0 C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	•	•	•	•	S	•
Acmltr A → CCR	TAP	06	2	1	$A \rightarrow CCR$	—	—	—	⑫	—	—
CCR → Acmltr A	TPA	07	2	1	$CCR \rightarrow A$	•	•	•	•	•	•

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of $N \oplus C$ after shift has occurred.

⑦ (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?

⑧ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?

⑨ (Bit N) Test: Result less than zero? (Bit 15 = 1)

⑩ (All) Load Condition Code Register from Stack. (See Special Operations)

⑪ (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

⑫ (ALL) Set according to the contents of Accumulator A.

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
=	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
−	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
		R	Reset Always
		S	Set Always
+	Boolean Inclusive OR;	↕	Test and set if true, cleared otherwise
⊕	Boolean Exclusive OR;	•	Not Affected
$\bar{M}$	Complement of M;	CCR	Condition Code Register
→	Transfer Into;	LS	Least Significant
0	Bit = Zero;	MS	Most Significant

DAA INSTRUCTION

Decimal Adjust Accumulator

A decimal digit may be represented as a 4 bit binary number e.g. 9 = 1001. Similarly a 2 digit decimal number can be represented by 8 bits, e.g. 49_{10} = 01001001. This form is known as Binary Coded Decimal or BCD, and is not to be interpreted as a normal binary number.

Addition of decimal numbers, expressed in BCD, is possible via the DAA (Decimal Adjust Accumulator) instruction as seen in this example:

```
LDA A  #$08
ADD A  #$06
DAA
```

The DAA instruction converts the normal hex sum, 0E, to 14, the expected decimal sum in BCD. This is accomplished internally by adding 6 in this example (0E + 06 = 14). Details of the internal operation of the DAA instruction are not essential to its use, but are given at the bottom of this page. What is important is that this instruction operates on ACC A, only after execution of the ADD, ADC or ABA instructions.

Assuming that symbolic addresses OLDATA and NUDATA each contain one BCD digit, write the instructions to produce the BCD sum in ACC A.

```
LDA A  OLDATA
ADD A  NUDATA
DAA
```

DAA Details: When two 2 digit BCD numbers are added a "carry", produced by the addition of the "least significant" column, sets the H bit of the CCR, e.g. 7 + 5 produces a carry and sets H, while 7 + 2 clears H. This H bit is added to to the "most significant" column, all operations being internal to the DAA instruction.

Decimal addition in BCD is equally valid for "2 digit" decimal data, e.g. $47_{10} + 78_{10}$. Here the BCD sum is 125, that is 25 plus a carry into the third column.

Write the instructions to add OLDATA and NUDATA, the sum going to TOTAL+1 and the carry going to TOTAL. Assume that OLDATA and NUDATA each contain 2 decimal digits in BCD form.

```
0100 7F 0150          CLR     TOTAL
0103 B6 0152          LDA A   OLDATA
0106 BB 0154          ADD A   NUDATA
0109 19               DAA
010A B7 0151          STA A   TOTAL+1
010D 24 03            BCC     FINI
010F 7C 0150          INC     TOTAL
0112            FINI
                      |
                      |
                      \
0150 0002     TOTAL   RMB     2
0152 0002     OLDATA  RMB     2
0154 0002     NUDATA  RMB     2
```

Lab instruments, such as digital voltmeters and frequency counters, often use BCD format to present data to a computer. Hence the DAA instruction vastly simplifies manipulation of this data, directly in BCD form.

Addition of "4 digit " decimal data also requires the detection of the carry bit after the 2 least significant columns are added. Use of the ADC (Add with Carry) instruction permits this carry to be added in when the next 2 most significant digits are added. Assume that OLDATA and NUDATA each contain 4 BCD digits in 2 bytes. Write the instructions to produce the 4 digit sum in the 2 bytes labelled TOTAL.

```
                *
                * ADDITION OF 4 CHAR BCD DATA.  SUM IN TOTAL.
                *
0100 7F 0150            CLR     TOTAL
0103 B6 0153            LDA A   OLDATA+1
0106 BB 0155            ADD A   NUDATA+1
0109 19                 DAA                 BCD SUM OF 2 LO DIGITS
010A B7 0151            STA A   TOTAL+1
010D B6 0152            LDA A   OLDATA
0110 B9 0154            ADC A   NUDATA
0113 19                 DAA                 BCD SUM OF 2 HI DIGITS
0114 B7 0150            STA A   TOTAL
                        |
                        |
                        |
```

This process could be extended to 6, 8 or N digit BCD addition. Note that the above program does not detect a carry beyond 4 digits; hence input should be limited to 3 BCD digits.

I N D E X