10.

Welcome back. I hope you've had a little fun with the final program in the last session. If you took the time to contrast the listing of that program with the previous one, you may have noticed a group of hexadecimal numbers and a series of USR routines in place of the BASIC POKEs. Remember that the synchronous address multiplexer — the SAM — uses write-only registers that are located in the upper area of memory. Fourteen of those addresses are used to set or reset the individual binary digits of a 7-bit video display address.

Coming into this lesson with concepts securely in your mind, you'll be solving a problem by structuring and programming a useful piece of software. Review comes first, then you'll get right into it.

* What does the BASIC POKE statement do?

It directly manipulates memory.

* What is the purpose of BASIC's PCLEAR statement?

To release memory for high-resolution graphics.

* What controls the video display address?

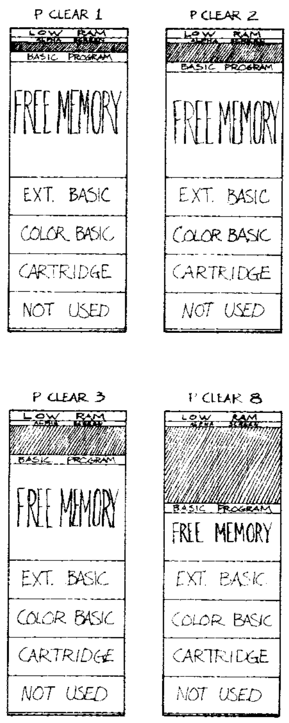
SAM registers.

* Where is the video screen located in the normal Color Computer?

At $0400.

* How is the address determined?

By writing to the SAM display offset registers.



Turn back to the last program listing. PCLEAR4 in the first line is intended to release memory for Extended BASIC's high-resolution graphics. What it actually does is move the BASIC program itself in memory, freeing a large block memory space between **$0600** and the start of the BASIC program. The way I've arranged the screens is first to print them on the screen, meaning they appear in memory at **$0400** to **$05FF**, the usual address of screen memory when you turn the computer on. The info is printed on the screen by seven subroutines, and then, byte by byte, POKEd into memory at **$0600**, **$0800**, **$0A00**, etc., in blocks of 512 bytes.

The screens are then prepared. All that remains is to redirect the video display by changing the video address in the SAM. My earlier program POKEd the changes in place, but the changes happen too slowly in BASIC. The results are illegible, with unwanted screens flickering by between POKEs. So I've set up some simple machine-language subroutines, which you can see in raw form in lines 9 through 15.

I'd like you to read these. Turn to your MC6809E data booklet, and open to pages 28 and 29. The first hexadecimal byte in the program is **$B7**. Look through the data booklet's numerical listing, and you find that **B7** corresponds to STA, or Store A Accumulator, in the extended addressing mode. The extended addressing mode, as you know, means that the two bytes following the opcode form an address where the data is loaded from or

* What is extended addressing?

An addressing mode where the two bytes following the opcode form an address where the data can be found.

* What addressing mode is utilized by STA $FFC7?

Extended addressing.

* Remember that it's the act of storing — not the information stored — into the SAM registers that determines the result. With that in mind, what action is taken by:
STA $FFC7
STA $FFC9
STA $FFCA
STA $FFCC
STA $FFCE
STA $FFD0
STA $FFD2

The video display offset address 0000011 is selected.

* What memory address is this?

$0600.

* The hex opcode for store A accumulator extended is $B7. What does $B7 06 00 indicate?

Store A accumulator at memory address $0600 (STA $0600).

* What is the clock speed of the Color Computer?

.89 MHz (894,886 clock cycles or pulses per second).

* How long is one clock cycle?

1.11746 microseconds (millionths of a second).

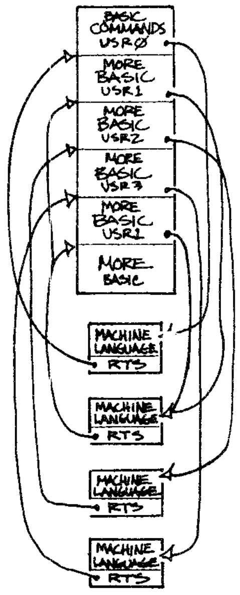
* How many clock cycles does a STA extended command take (the information is in the data booklet).

5 clock cycles.

stored. The next two hex numbers in the program are **$FF** and **$C7**. Your SAM data booklet will tell you that **$FFC7** is the address to set the least-significant bit of the video display address.

Follow the remaining hex bytes in the listing. You'll see **B7 FF C9**, meaning Store A Accumulator at **$FFC9**; **B7 FF CA**, Store A Accumulator at **$FFCA**; **B7 FF CC**, Store A Accumulator at **$FFCC**; and **39**. Check **$39** in the numerical instruction list on page 28 of the MC6809E data booklet. It's an opcode that will become very familiar — it is **RTS**, Return from Subroutine.

So the first group of bytes in line 9 of the BASIC program store the A Accumulator at **$FFC7**, **$FFC9**, **$FFCA** and **$FFCC**. A check of the SAM registers will show that these actions will place the binary value **0011** in bits 9, 10, 11 and 12 of the video address. Bits 13, 14, and 15 (the most signficant bits) are all zero, because that's where they were established when the computer was turned on. The full result of this short subroutine, then is to create the video address **0000 0110 0000 0000**. I'll translate that for you. It's address **$0600**, the address of the first screen the BASIC program POKEd into memory. By analyzing each of lines 9 through 15, you will see that the video display addresses created are **$0600**, **$0800**, **$0A00**, and so forth.



These seven short machine-language subroutines, then, are a quick version of the BASIC POKEs that were used to redirect the screen in the previous program. The speed here, however, is too fast to see. How fast is it? Glad I asked that. Flip to page 31 in the MC6809E data booklet, and look up the mnemonic **STA**. Under the heading "Extended", you'll find the opcode **$B7**. The next column tells you that a Store A Accumulator Extended takes five clock cycles. There are four Store A Accumulator instructions in each video display switching subroutine, meaning a total of 20 clock cycles. The **RTS** (Return from Subroutine) takes 5 clock cycles. The whole subroutine takes 25 clock cycles. At your Color Computer clock rate of 894,886 clock cycles per second, that means the subroutine is finished with its work in .00002794 seconds — 30 millionths of a second, about the time it takes the electron beam to sweep halfway across the TV screen.

I want to close a knowledge gap now. Obviously I've been talking about machine language subroutines in this BASIC program. BASIC puts those subroutines into memory in a very clumsy way. Look at the program listing. In lines 9 through 15 are a series of BASIC DATA statements in which the hexadecimal numbers are treated as strings. In line 16, I have variable X select the memory area to be used; in this case it's 16293 to 16383, hexadecimal addresses **$3FA5** to **$3FFF**.

The next step has the hexadecimal byte masquerading as a two-character ASCII string read as variable A$. BASIC identifies hexadecimal by the symbol "&H", so "&H" is

LET A$= "C3"
LET B=
""&H""+A$
(B$ BECOMES &HC3)
VAL(B$) is VAL(&HC3)
VAL(B$)=
195

concatenated with each two-character ASCII string. In this way, BASIC can be tricked into taking the value of the string, and that value can then be POKEd into memory. All that happens in line 16. Seven machine-language subroutine entry points are established in lines 17 through 23. Extended Color BASIC allows ten entry points altogether named USR0 through USR9; this program defines USR1 through USR7 for the seven screens to be displayed. Finally, lines 24 through 41 execute these subroutines in a fancy series of FOR-NEXT loops, and delay appropriately. By changing the order of the loops, you can make the seven messages flicker and flash in a variety of ways.

USRØ - USR9
at
POWER UP

USRØ	$B44A
USR1	$B44A
USR2	$B44A
USR3	$B44A
USR4	$B44A
USR5	$B44A
USR6	$B44A
USR7	$B44A
USR8	$B44A
USR9	$B44A

Here's a recap: Seven 512-byte screens are created in the memory below the BASIC program, allocated by PCLEAR4. These screens are displayed by machine-language subroutines that switch the video display registers in the SAM. I hope this hybrid BASIC / machine-language program gives you some ideas for effective but simple program displays.

DEFUSR1=
16293

USR Ø	$B44A
USR 1	$3FA5
USR 2	$B44A
USR 3	$B44A
USR 4	$B44A
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A

As for the knowledge gap, the technique for creating short machine-language programs and POKEing them into memory via BASIC is something you can use often. Write the program, either byte-by-byte or using an editor/ assembler. Take the hexadecimal opcodes and operands in the order they will appear in memory, and put the values into a bunch of BASIC DATA statements. Read each value, convert it to a number BASIC can use, and POKE it into memory. By using the DEFUSR command, define where your program will begin execution. From that point on, it only takes a USR command to execute your machine-language program. Review the program you've just run until you understand how that's done.

DEFUSR 2=
16306

USRØ	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$B44A
USR4	$B44A
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A

Before I leave this program, please load the mnemonic source code that follows.

Program #17, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program is not similar to the listing, or if an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
3FA5                00100         ORG    $3FA5
3FA5 B7    FFC7     00110 SCRN1   STA    $FFC7
3FA8 B7    FFC9     00120         STA    $FFC9
3FAB B7    FFCA     00130         STA    $FFCA
3FAE B7    FFCC     00140         STA    $FFCC
3FB1 39             00150         RTS
3FB2 B7    FFC6     00160 SCRN2   STA    $FFC6
3FB5 B7    FFC8     00170         STA    $FFC8
3FB8 B7    FFCB     00180         STA    $FFCB
3FBB B7    FFCC     00190         STA    $FFCC
3FBE 39             00200         RTS
3FBF B7    FFC7     00210 SCRN3   STA    $FFC7
3FC2 B7    FFC8     00220         STA    $FFC8
```

* How long is that?

5 times 1.11746, or 5.5873 microseconds.

* How many STA extendeds is that per second?

1000000 microseconds divided by 5.5873 per STA extended instruction, or roughly 179,000 per second.

* BASIC can perform roughly 68 POKEs per second. How much faster is the machine language equivalent of STA extended?

179,000 divided by 68, or about 2,632 times faster.

* What is the standard symbol for hexadecimal?

The dollar sign ($).

* What is the BASIC symbol for hexadecimal?

The symbol ampsersand plus the letter H (&H).

* What command is used for a BASIC machine language entry point?

USR.

* BASIC needs to know the starting point of a machine language program. How does it get it?

With the DEFUSR command.

* What does DEFUSR3=&H3FBF mean?

It means that the entry point (execution address) for USR routine number 3 is at location $3FBF.

* Write a statement that informs BASIC that machine language program #7 begins at $3FF3.

DEFUSR7=&H3FF3.

* What is BASIC's representation of hexadecimal?

Ampersand plus H (&H).

* If variable C$ is A9, write a statement to make C equal to the hexadecimal value of C$.

C = VAL("&H"+C$)

* What is hand assembly?

Figuring the hex (binary) code byte by byte from the mnemonic (source) code.

* Hand assemble STA $FFC7 into hex, and then binary, code.

STA $FFC7 becomes $B7 FF C7, which becomes 10110111 11111111 11000111.

* What addressing mode is this?

Extended addressing.

* How many bits represent an address?

16 bits.

* How many hexadecimal characters is this?

4 hex characters.

* How many bits represent the memory contents at an address (the data)?

8 bits.

* How many hex characters is this?

2 hex characters.

* What is the value 00010010 in hexadecimal?

$12

* What are the ASCII values for "1" and "2"?

$31 and $32.

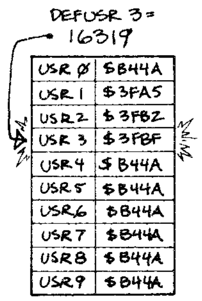
```
3FC5 B7    FFCB      00230          STA     $FFCB
3FC8 B7    FFCC      00240          STA     $FFCC
3FCB 39              00250          RTS
3FCC B7    FFC6      00260 SCRN4    STA     $FFC6
3FCF B7    FFC9      00270          STA     $FFC9
3FD2 B7    FFCB      00280          STA     $FFCB
3FD5 B7    FFCC      00290          STA     $FFCC
3FD8 39              00300          RTS
3FD9 B7    FFC7      00310 SCRN5    STA     $FFC7
3FDC B7    FFC9      00320          STA     $FFC9
3FDF B7    FFCB      00330          STA     $FFCB
3FE2 B7    FFCC      00340          STA     $FFCC
3FE5 39              00350          RTS
3FE6 B7    FFC6      00360 SCRN6    STA     $FFC6
3FE9 B7    FFC8      00370          STA     $FFC8
3FEC B7    FFCA      00380          STA     $FFCA
3FEF B7    FFCD      00390          STA     $FFCD
3FF2 39              00400          RTS
3FF3 B7    FFC7      00410 SCRN7    STA     $FFC7
3FF6 B7    FFC8      00420          STA     $FFC8
3FF9 B7    FFCA      00430          STA     $FFCA
3FFC B7    FFCD      00440          STA     $FFCD
3FFF 39              00450          RTS
           0000      00460          END
00000 TOTAL ERRORS
SCRN1    3FA5
SCRN2    3FB2
SCRN3    3FBF
SCRN4    3FCC
SCRN5    3FD9
SCRN6    3FE6
SCRN7    3FF3
```

Type A/NO and hit <ENTER>. Lines of information scroll by. The incredible thing about this mnemonic source code — and most mnemonic source code — is that it looks so massive. Here are 36 lines of typing, 7 labels, 8 columns wide, practically filling a page. And yet all this resolves into a mere 91 bytes of actual program, little more than a third of what a BASIC program line can hold.

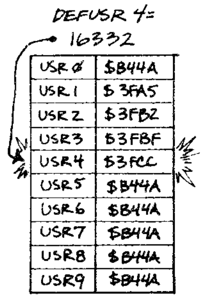
Since I knew precisely what I wanted, and since this program was so short and consistent, I actually figured out the hex code byte by byte using the MC6809E data booklet. Later I typed this source code for you. But in doing the hand programming, I had to keep track of where each subroutine began. The nice part about an editor/assembler is that whatever you have in mind can be typed and examined easily, even if it seems long. The editor/assembler picks up typing errors, whereas hand assembling each byte can be a highly error-prone procedure. Plus, by liberally scattering labels in the code, critical addresses can be identified; in fact, the assembler provides a complete display of all labels at the end of the assembled listing. Which teaches you more? My vote is for hand assembly. I'll help you with some of that.

For hand assembly you'll need paper and pencil, plus your MC6809E data booklet open to pages 30 and 31. The problem will turn away from flashy video displays for awhile; here it is:

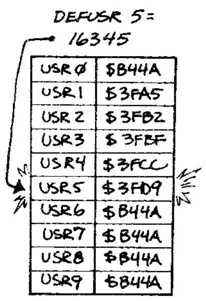
Given an address transferred from a BASIC program, create a display which will present eight lines of information. The first line will contain the address and eight hexadecimal bytes of memory contents separated by spaces. If the address is **$2000**, for example, the display


DEFUSR 3=
16319

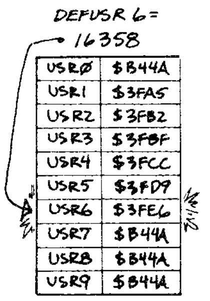
USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$B44A
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A


DEFUSR 4=
16332

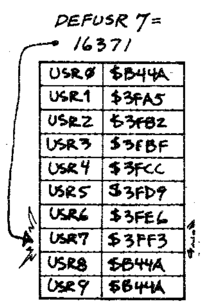
USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$3FCC
USR 5	$B44A
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A


DEFUSR 5=
16345

USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$3FCC
USR 5	$3FD9
USR 6	$B44A
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A

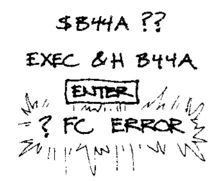

DEFUSR 6=
16358

USR Ø	$B44A
USR 1	$3FA5
USR 2	$3FB2
USR 3	$3FBF
USR 4	$3FCC
USR 5	$3FD9
USR 6	$3FE6
USR 7	$B44A
USR 8	$B44A
USR 9	$B44A



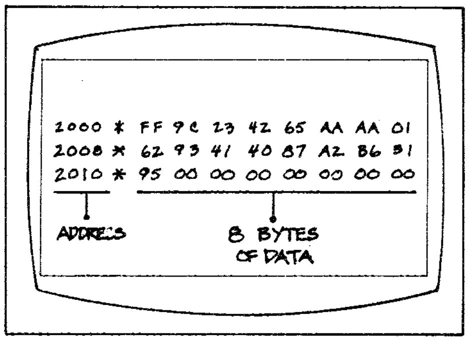
DEFUSR 7=
16371

USRØ	$B44A
USR1	$3FA5
USR2	$3FB2
USR3	$3FBF
USR4	$3FCC
USR5	$3FD9
USR6	$3FE6
USR7	$3FF3
USR8	$B44A
USR9	$B44A



$B44A ??
EXEC &H B44A
ENTER
? FC ERROR

should print **$2000** followed by the data found in memory locations **$2000**, **$2001**, **$2002**, etc., up to **$2007**. In the next line, the address **$2008** would be displayed, together with the memory data found at **$2008** through **$200F**. And on down for a total of eight lines. Ready?



```
2000 * FF 9C 23 42 65 AA AA 01
2008 * 62 93 41 40 87 A2 B6 51
2010 * 95 00 00 00 00 00 00 00
```

Know how to tell if you're ready? Think about Session 8, where I presented a dozen machine language instructions and showed how they worked, including how flags were affected. If that's not clear and reasonably fresh in your mind, review it now. When those instructions make sense to you, you're ready to move on.

The problem at hand is to transfer an integer from BASIC which represents an address in memory you'd like to examine. That examination will display 8 lines, each line containing one address and 8 consecutive bytes of memory data. In all, 64 bytes of data will be displayed. First, conceptualize the problem. Information in integer form is to be transferred to the machine-language program. That part is easy; the USR function is used, with the target address being the operand in parentheses. You've already used the integer-conversion routine from the BASIC ROM in order to retrieve a value from BASIC for your machine-language program's use, so that's easy.

Once you've got the integer value in your own program, two things need to be done. First, it has to be treated as displayable information. The address must be converted to four ASCII characters for presentation as a hexadecimal display. Second, the integer has to be treated as the address itself in order to retrieve the memory information for display.

How about an integer-to-ASCII conversion routine, then? You'll want to break it down into simple modules, if possible. Start by looking for modularity, small consistent units that you can program. What you know you have are 16 binary digits which you want to represent on the screen as four ASCII characters in hexadecimal notation. There's a clue there. 16 binary digits. Four ASCII characters. You already know that a single hexadecimal number represents four binary digits. The solution lies in that knowledge: treat each four-bit group as an identical task. A single subroutine.

* What is the value of 11001101 in hexadecimal?

$CD

* What are the ASCII values for "C" and "D"?

$43 and $44.

* What is the value 10001110 in hexadecimal?

$8E

* What are the ASCII values for "8" and "E"?

$38 and $45.

* An address is $A0D7. What are the four ASCII values (A, 0, D and 7)?

$41, $30, $44 and $37.

* What is the ASCII value for a space?

$20.

* To display the address $A0D7, a space, and the contents of $A0D7 (which is $8E), what ASCII values must be used?

$41 30 44 37 20 38 45

* Where are these ASCII values placed?

In display memory.

* Where is display memory on the normal Color Computer?

From $0400 to $05FF.

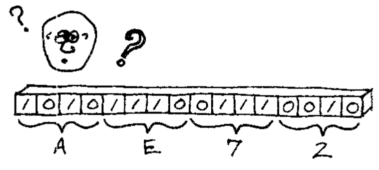
* How many bytes is the value $8E?

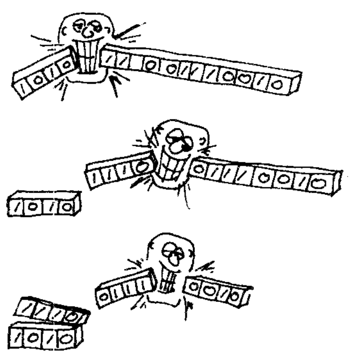
One byte, $8E.

* How many bytes are the ASCII values needed to represent the value $8E?

Two bytes, $38 and $45.

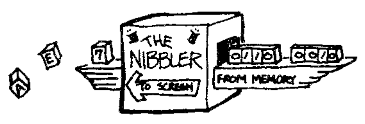
That line of thinking brings you one step closer to a modular approach. Each time you have four bits in hand, you can call the subroutine that creates an ASCII character from them. Now you need only sketch out that subroutine. Recall a few sessions ago how, in order to access a table of encrypted codes, a constant value had to be subtracted from the ASCII characters to obtain numbers starting from zero. In this case, you have a complementary situation. You have four binary digits equivalent to the hexadecimal numbers **0** through **F**. In order to produce ASCII characters, then, it's necessary to *add* a constant value. To display the number zero as the *character* 0 with the ASCII value of hex **$30**, you would add hex **$30**. To display the number one as the character 1 with the ASCII value **$31**, again you would add **$30**. You would do that right up through number nine which is displayed as the character 9, ASCII value **$39**. The constant you add is **$30**.



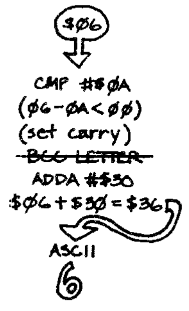


So far so good. But when you get to number A, you're in a little trouble. Binary **1010** is number **A**. Character A is ASCII value hex **$41**. The constant you must add to number A to get character A is hex **$37**. It's consistent from A through F — add **$37** to the value and you get the ASCII character.

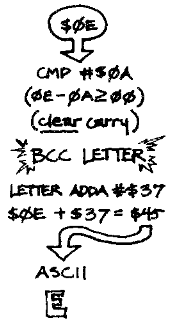
How do you reconcile the two different constants? The answer is simple: you don't. You find out whether the value is **0** through **9** or **A** through **F**, and add the constant **$30** or **$37** accordingly.



That looks like enough information for a subroutine. The "entry condition", as it's called, is a group of four binary digits. That four-bit number is checked to see whether it is greater or less than 9. If it's greater than 9, you add the constant **$37**; if it's 9 or less, you add the constant **$30**. The result is an ASCII character which, when displayed, represents the hexadecimal numerical value. The ASCII character is the subroutine's "exit condition". The nice part about a subroutine like this is its versatility — not only can it be used to display the digits of an address, it's just as good for displaying the bytes of memory data.



Mnemonically speaking, that would operate like this. The A Accumulator enters with the four-bit number. It's compared immediate with **$0A**. If the number is greater than nine, the carry/borrow flag would not be set. The program would Branch on Carry Clear to an instruction to add **$37** and then return from subroutine; otherwise it would add **$30** and return from subroutine. The A Accumulator enters with the number and exits with the ASCII character. Pretty slick.



It would look like this, assuming the A Accumulator holds the four-bit number:

```
CONVRT  CMPA    #$ØA
        BCC     LETTER
        ADDA    #$3Ø
        RTS
LETTER  ADDA    #$37
        RTS
```

* How many bytes is the address $A0D7?

Two bytes, $A0 and $D7.

* How many bytes are the ASCII values needed to represent the value $A0D7?

Four bytes, $41, $30, $44 and $37.

* What are the ASCII values for the characters "0" through "9"?

$30 through $39.

* What are the ASCII values for the characters "A" through "F"?

$41 through $46.

* What is the number $8E in binary?

$8E in binary is 1000 1110.

* In the number $8E, which bits represent the number 8?

The leftmost four bits.

* In the number $8E, which bits represent the number E?

The rightmost four bits.

* What are the leftmost and rightmost four bits of $8E?

1000 and 1110

* What are the binary values for 8 and E?

0000 1000 and 0000 1110.

* What are the binary values for ASCII "8" and ASCII "E"?

0011 1000 and 0100 0101.

* What is the difference between binary 8 and ASCII "8"?

Binary 8 is 0000 1000 and ASCII "8" is 0011 1000; the difference is 0011 0000, or $30.

Now there's the task of breaking the 16-bit address into four 4-bit groups. Half of that's done already, since the 16-bit address is split into two 8-bit bytes. Creating this subroutine from there demands just a little convoluted thinking.

You have 8 bits. You only want to use four bits at a time, and these four bits have to be in the least-significant positions. In other words, if the number is **$3C**, you want to convert the four bits **0011** into a **3**, and the four bits **1100** into a **C**. The least-signficant four bits of the byte are just about ready to use. All that remains is to temporarily get rid of the most-significant four bits. The term is "mask" the bits, meaning create a mask so that only the bits you need show through.

The mask here is AND. Recall how the AND instruction works. Both conditions must be a one for the result to be a one. To mask out the four leftmost bits of the byte, then, you would AND each of those four bits with zero. To mask IN the four rightmost bits you would AND each of those four bits to one. I'll repeat that a different way. If the leftmost four bits are ANDed with zero, no matter what those bits are, the result of the ANDing will be zero. If the rightmost four bits and ANDed with one, no matter what those bits are, they will effectively remain the same.

$3C=
00111100
AND 00001111
00001100
=$ØC

Scratch it out on paper and look at it. Use the example **$3C** that I just mentioned. Write down the binary equivalent: **0011 1100**. Underneath it, write down the mask: **0000 1111**. Now use the AND function:

```
Ø AND Ø is Ø
Ø AND Ø is Ø
1 AND Ø is Ø
1 AND Ø is Ø
```

That's the leftmost four bits. Now the rightmost:

```
1 AND 1 is 1
1 AND 1 is 1
Ø AND 1 is Ø
Ø AND 1 is Ø
```

There are the rightmost four bits. The mask to use here is **$OF**. To recap: to retrieve the least-signficant four bits of a byte, use the mask **$OF**.

You can pause here to review that section if you like.

The next task is to retrieve the leftmost four bits. If logic holds, then you can again use a mask. Since the bits you want are to the left, then the mask **1111 0000** should suffice. That's **$FO**; it will result in the four leftmost bits being masked *in*, and the four rightmost bits being masked *out*.

There's a problem, though. Although it masks in the bits you want, they're not in the correct place. You need them on the *right* side of the byte to represent the 4-bit numbers **$O** through **$F**. You have to get those bits from left to right.

*** What is the difference between binary E and ASCII "E"?**

Binary E is 0000 1110 and ASCII "E" is 0100 0101; the difference is 0100 0000, or $37.

*** What is the constant difference between binary values 0 through 9 and ASCII values "0" through "9"?**

The constant difference is $30.

*** What is the constant difference between binary values A through F and ASCII values "A" through "F"?**

The constant difference is $37.

*** What logical function states: both of two conditions must be true for the result to be true?**

The AND function.

*** How are the rightmost four bits retrieved from the number $0E (1000 1110)?**

By masking the leftmost four bits.

*** What mask is used?**

AND 00001111.

*** If A contains $0E, what mnemonic command is used to retrieve the rightmost four bits?**

ANDA #$0F (AND A accumulator immediate with $0F, binary 00001111).

*** What constant is added to $0E to produce the ASCII character "E"?**

$37.

*** How are the leftmost four bits retrieved from the number $8E (1000 1110)?**

By shifting the bits right four times.

*** When $8E is shifted right once, what is the result (in hex and binary)?**

0100 0111 ($47).

*** When $8E is shifted right twice, three times, and four times, what are the results (in hex and binary)?**

0010 0011 ($23), 0001 0001 ($11) and 0000 1000 ($08).

*** What constant is added to $08 to produce the ASCII character "8"?**

$30.

*** What is necessary to convert the least significant half of a byte to a 4-bit number?**

Masking with $0F.

*** What is necessary to convert the most significant half of a byte to a 4-bit number?**

Rotating right four times.

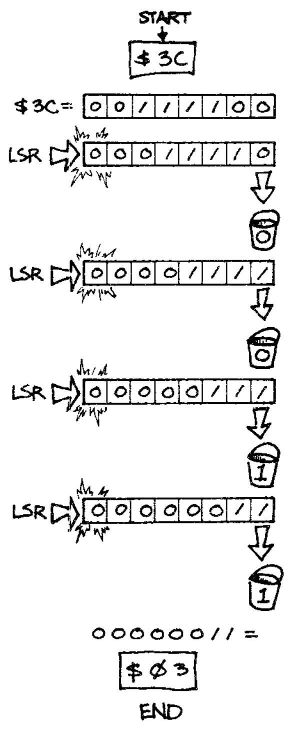
*** What is necessary to convert a 4-bit binary number to a hexadecimal ASCII character?**

The addition of a constant.

Recall the various rotate and shift commands from an earlier session. You'll need to refer to your MC6809E data sheet to choose the particular rotate or shift you want; open to pages 30 and 31.

You know that you need to move these bits to the right. Your choices are ASR (arithmetic shift right), LSR (logical shift right), and ROR (rotate right). Look at each one. ASR reproduces the leftmost bit each time you shift, so this doesn't look very good. If you shifted first and masked second, it would work. How about LSR? It shifts right and brings zeros in from the left as it shifts. That one looks good. Finally, ROR swings the bits 'round from the other side of the byte, so you would need to mask the results afterward.

The logical shift right (LSR) looks the best. In fact, it looks excellent. Since the bits shifted out the right side end up in the bit bucket, and zeros come in from the left, you don't even have to bother masking this before you use it. The process of shifting it right gives you not only the four bits you need, but eliminates those you don't want.



Here's a summary of these two program segments: the byte is to be displayed as two hexadecimal ASCII characters. The leftmost four bits are obtained by logically shifting the byte right four times. The rightmost four bits are obtained by masking the original byte with **$0F**. All that remains is to make sure the original value is saved before modifying it. Push A Accumulator will take care of saving the byte, and Pull A Accumulator will get it back when it's needed. In terms of mnemonics, and assuming the value to be displayed is in the A Accumulator, the complete routine would look like this:

```
BYTBIT  PSHS   A       Push A Accumulator onto stack
        LSRA           Logical Shift Right A Accumulator
        LSRA           Logical Shift Right A Accumulator
        LSRA           Logical Shift Right A Accumulator
        LSRA           Logical Shift Right A Accumulator
        JSR    CONVRT  Jump to ASCII conversion subroutine
        JSR    DISPLY  Jump to screen display subroutine
        PULS   A       Pull A Accumulator from stack
        ANDA   #$ØF    AND A Accumulator immediate with $ØF
        JSR    CONVRT  Jump to ASCII conversion subroutine
        JSR    DISPLY  Jump to screen display subroutine
```

At this point, two major portions of the problem have been solved: the 8-bit byte has been converted to two 4-bit numbers, and those 4-bit numbers have been converted to ASCII characters. The screen display routine has yet to be done. I'll leave you with these considerations: your program has to know where to start the screen display in memory, that is, it has to be initialized. The current screen display position has to be updated so that the next character displayed will appear in the next available position.

Review this lesson, and consider those problems for next time.

11.

The topic is hand assembly. Last time I started you working on a program to display memory locations and their contents. At the end of the session, you had produced two pieces of that program: the byte-to-nybble conversion routine (a nybble is four bits), and the hexadecimal-to-ASCII conversion routine. The byte-to-nybble conversion was made up of two steps. To move the most-significant nybble into the righthand portion of the byte, the byte was logically shifted right four times. To obtain the least-significant nybble, a mask of **$0F** was ANDed with the value of the byte.

The problem I posed at the end of the session was this one: create a single-character display subroutine that, when called, places a character in the correct location on the screen and updates the program to point to the next available screen location.

To help solve this, I hope you thought back to the message-display program you created in the third session. There wasn't much to that display routine, and there isn't much to this one either. At the beginning of this program, then, you would initialize the first screen location, perhaps in the Y register. Each Color Computer screen line is 32 characters long — that's hex **$20**. So to start on the fourth line of the screen, you would load the Y register with the immediate value of **$0480** at the start of the program:

```
          LDY     #$0480
```

is the mnemonic. If the ASCII value to be displayed is the A Accumulator, and the Y register points to the current location on the screen, then you would store the A Accumulator in memory — display memory, that is — indexed by Y. To update that location, choose the auto-increment/decrement zero-offset indexed mode. You remember that mouthful. That's Store A Accumulator at memory indexed simply by Y, auto-increment Y by one, and then return from subroutine. Label it DISPLY:

```
DISPLY  STA   ,Y+
        RTS
```

Hand assembly really hasn't gotten underway yet. At this point, the program is still being structured and converted into mnemonic source code. So far, a complete byte-to-ASCII conversion system has been developed. What's to come is a display routine, plus a kind of executive structure.

*** What is the location of the normal display screen on the Color Computer?**

$0400 to $05FF.

*** Each line of the display is 32 characters long. What line starts at $0480?**

If $0400 is the start of the first line, then $0480 is the start of the fifth line.

*** If the Y register points to screen location $0480 and the A register contains the ASCII value, what mnemonic instruction would place the ASCII value on the screen?**

STA ,Y

*** What mnemonic instruction would place the ASCII value on the screen, and automatically move the Y pointer register to the next screen position?**

STA ,Y+

*** Write two instructions that, given the conditions just used, create a complete ASCII display and screen update routine.**

```
STA  ,Y+
RTS
```

That should do the trick. A short, sweet 3-byte subroutine that illustrates the power of the 6809 processor.

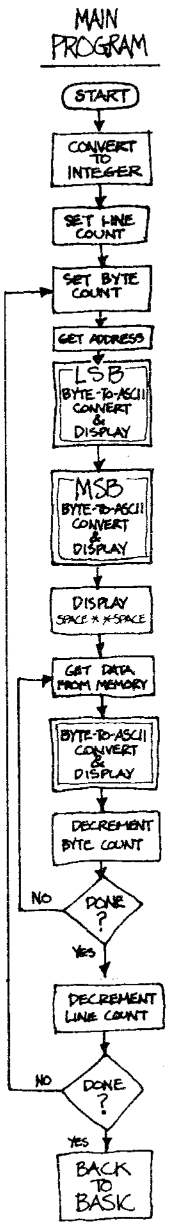
That seems to cover the necessary subroutines — conversion and display. What's left to create is a kind of executive program which accepts the address from BASIC, searches for the memory data, and calls the subroutines you've just created. This executive's job would be to call for the value from BASIC, initialize the screen parameters, do the screen line and screen character counting, call the convert and display subroutines, and return to BASIC when all is done.

The sequence as I see it comes out to 15 steps:

1. Get the target address from BASIC
2. Initialize the screen starting position
3. Initialize the line and character counts — 8 lines, memory bytes per line
4. Convert and display the most-significant byte of the memory address
5. Convert and display the least-significant byte of the memory address
6. Display a space as a separator
7. Display two stars or other separators
8. Display another space as another separator
9. Get the memory contents of the address
10. Convert and display that memory byte
11. Display another space as a divider
12. Increment the target address
13. Loop for 7 more memory bytes, for a total of 8
14. Loop for 7 more lines of address, for a total of 8
15. And finally, return to BASIC

I've prepared a program that follows these steps; open to your documentation and follow along. The program is in mnemonics, which you will be hand-assembling. I'll explain each line briefly; those which you haven't already written should fall into place.

```
-----------------------------------------------------
        JSR    $B3ED        BASIC INTEGER-CONVERT ROUTINE
        LDY    #$Ø48Ø       FIRST SCREEN LOCATION TO USE
        TFR    D,X          GIVE INT-CONV RESULT TO X REG
        LDA    #8           PUT 8 LINES INTO ACCUMULATOR
        STA    <ØØØ1        LINE COUNT INTO DIR. PAGE Ø1
.....................................................
LLOOP   LDA    #8           PUT 8 BYTES INTO ACCUMULATOR
        STA    <ØØØØ        BYTE COUNT INTO DIR. PAGE ØØ
        TFR    X,D          INT-CONV RESULT BACK TO D REG
        JSR    BYTBIT       BYTE-TQ-ASCII CONV. & DISPLAY
        TFR    B,A          MOST SIGN. BYTE INTO A ACCUM.
        JSR    BYTBIT       BYTE-TO-ASCII CONV. & DISPLAY
        LDA    #$2A         PUT ASCII FOR "*" INTO A ACC.
        LDB    #$2Ø         ASCII FOR SPACE INTO B ACCUM.
        STB    ,Y+          DISPLAY SPACE, GET NEXT POSN.
        STA    ,Y+          DISPLAY STAR, GET NEXT POSN.
        STA    ,Y+          DISPLAY STAR, GET NEXT POSN.
        STB    ,Y+          DISPLAY SPACE, GET NEXT POSN.
```



* What does STA ,Y+ mean?

Store A accumulator to memory indexed by the Y register, with no offset, and automatically increment Y.

* Given that A contains $2A and B contains $20, what do the following four instructions do?
STB ,Y+
STA ,Y+
STA ,Y+
STB ,Y+

The four instructions display space, star, star, space.

* What does JSR $B3ED identify on the Color Computer?

An integer conversion subroutine in the BASIC ROM.

* What are the results of JSR $B3ED?

A 16-bit signed integer is found in the D register.

* What does integer mean?

A number without a fractional (or decimal) part; a whole number.

* What does "signed integer" mean?

It means the number is positive or negative.

* How is the sign indicated?

By the leftmost bit; 0 is positive, 1 is negative.

* In the display program, how is the sign information used?

It isn't. The number is treated as a 16-bit unsigned integer.

* In the program, the instruction STA <$0001 appears. What addressing mode is this?

Direct addressing.

* In the program, the instruction LDA #$2A appears. What addressing mode is this?

Immediate addressing.

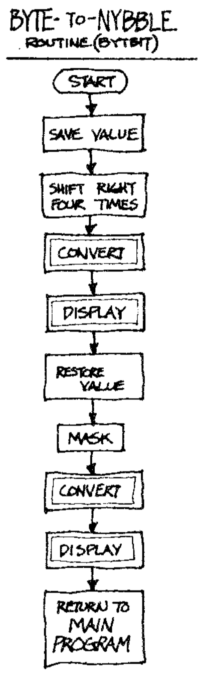
* In the program, the instruction JSR $B3ED appears. What addressing mode is this?

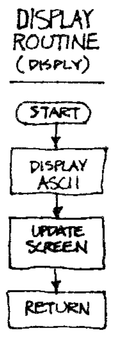
Extended addressing.

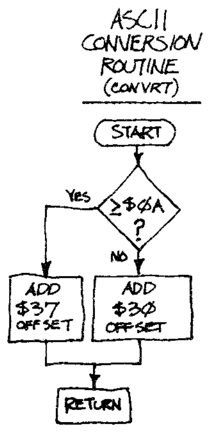
* In the program, the instruction BNE LLOOP appears. What does BNE LLOOP mean?

It means Branch Not Equal to the instruction labeled in the source listing "LLOOP".

```
BLOOP   LDA    ,X+          GET MEMORY CONTENTS X-INDEXED
        JSR    BYTBIT       BYTE-TO-ASCII CONV. & DISPLAY
        STB    ,Y+          DISPLAY SPACE, GET NEXT POSN.
        DEC    <ØØØ1        DECREMENT NUMBER OF BYTES
        BNE    BLOOP        REPEAT UNTIL ALL 8 DISPLAYED
        DEC    <ØØØØ        DEC. NUMBER OF DISPLAY LINES
        BNE    LLOOP        REPEAT UNTIL ALL 8 DISPLAYED
        RTS                 BACK TO BASIC WHEN ALL DONE
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
BYTBIT  PSHS   A            SAVE BYTE STORED IN A ACCUM.
        LSRA                SHIFT TO RIGHT ONE BIT  ....
        LSRA                .... AND SHIFT ONE MORE ....
        LSRA                .... AND SHIFT ONE MORE ....
        LSRA                .... TIL 4 BITS ARE AT RIGHT
        JSR    CONVRT       NYBBLE-TO-ASCII CONVERSION
        JSR    DISPLY       DISPLAY ASCII CHAR. & UPDATE
        PULS   A            RECOVER ORIGINAL BYTE STORED
        ANDA   #$ØF         MASK IN RIGHT-HAND NYBBLE
        JSR    CONVRT       NYBBLE-TO-ASCII CONVERSION
        JSR    DISPLY       DISPLAY ASCII CHAR. & UPDATE
        RTS                 TWO CHARS. CONV'D & DIPLAYED
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
CONVRT  CMPA   #$ØA         COMPARE NYBBLE AGAINST $ØA
        BCC    LETTER       IF CARRY CLEAR, A ACC. >= $ØA
        ADDA   #$3Ø         ELSE IS A NUMBER, SO ADD $3Ø
        RTS                 CONVERSION COMPLETE; RETURN
        . . . . . . . . . . . . . . . . . . . . . . . . . . .
LETTER  ADDA   #$37         IT IS A LETTER, SO ADD $37
        RTS                 CONVERSION COMPLETE; RETURN
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
DISPLY  STA    ,Y+          DISPLAY ASCII, UPDATE SCREEN
        RTS                 DIPLAYED & UPDATED; RETURN
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```







Now comes the time-consuming part. I want you to translate each one of these mnemonics into the binary opcodes and operands the computer will need to execute the program. I'm confident this program works — there are some anomalies, but you'll discover them soon enough — so open your MC6809E data booklet to pages 30 through 33.

Assume that the program will be stored in memory beginning at **$3F00**. Since some of you have 16K machines whose uppermost RAM address is **$3FFF**, this gives you 256 bytes of room for the program. I can tell you now that this program will occupy less than 100 bytes, and with some experience you'll be able to scope out program lengths like this one. One other assumption to make is the address of the Direct Page, which is **$00**; that information is provided in your EDTASM+ manual, in the memory map appendix, which also informs you that direct page addresses **$00** through **$7F** are free for your use.

For the hand assembly, you'll need several sheets of lined notebook paper, with the addresses **$3F00** through **$3F60** in a column down the left side. This is a good time to take a break for a review, and also to get the paper ready.

* What addressing mode is BNE LLOOP?

Relative addressing.

* In the program, the instruction STB ,Y+ appears. What addressing mode is this?

Indexed addressing (specifically, zero-offset indexed).

* In the program, the instruction LSRA appears. What addressing mode is this?

Inherent addressing.

* What is hand assembly?

Figuring the hex (binary) code byte by byte from the mnemonic (source) code.

* The following inherent instructions appear in the program. Hand assemble each:

* Hand assemble LSRA.

$44

* Hand assemble RTS.

$39

* The following immediate instructions appear in the program. Hand assemble each one:

* Hand assemble LDY #$0400.

$10 8E 04 00

* Hand assemble LDA #$08.

$86 08

* Hand assemble LDB #$20.

$C6 20

* Hand assemble ANDA #$0F.

$84 0F

* Hand assemble ADDA #$30.

$8B 30

* The direct instruction STA <$0001 appears in the program. Hand assemble it.

$97 01

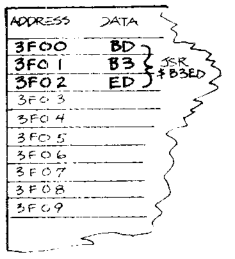
* The following register instructions appear in the program. Hand assemble each one:

* Hand assemble TFR D,X.

$1F 01

```
* Hand assemble TFR B,A.

$1F 98

* Hand assemble PSHS A.

$34 02

* Hand assemble PULS A.

$35 02

* The following indexed
instructions appear in the
program. Hand assemble each
one:

* Hand assemble STB ,Y+

$E7 A0

* Hand assemble STA ,Y+

$A7 A0

* Hand assemble LDA ,X+

$A6 80

* The following immediate
instructions do not appear in
the program. Hand assemble each
one.

* Hand assemble ADDD #$C3C3

$C3 C3 C3

* Hand assemble ANDCC #$AF

$1C AF

* Hand assemble CMPX #$05FF

$8C 05 FF

* Hand assemble CMPA #$FF

$81 FF

* Hand assemble EORA #$20

$88 20

* Hand assemble LDD #$BBAA

$CC BB AA

* Hand assemble ORB #$AC

$CA AC

* Hand assemble SUBA #$02

$80 02

* The following extended
instructions do not appear in
the program. Hand assemble each
one.

* Hand assemble ADDA $1000

$BB 10 00
```

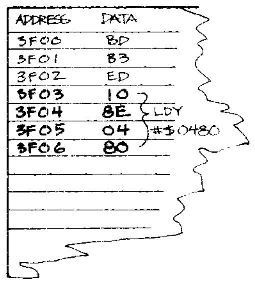
You should have your notebook paper ready, and your MC6809E data booklet open to page 30.



ADDRESS	DATA	
3F00	BD	JSR $B3ED
3F01	B3	
3F02	ED	
3F03		
3F04		
3F05		
3F06		
3F07		
3F08		
3F09		

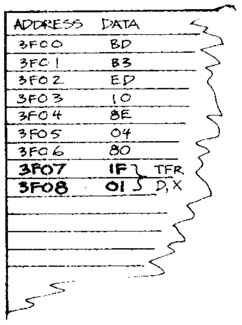
Start with the first instruction, JSR **$B3ED**. Find JSR on page 30. This is an extended addressing mode; the opcode you should find is **$BD**. On your paper, next to address **$3F00**, write **$BD**. At address **$3F01**, write the first byte of the operand, which is **$B3**. At address **$3F02**, write the second byte, **$ED**. You have hand-assembled the first instruction, **JSR $B3ED**, into three binary bytes, **$BD B3 ED**.

Your pencil should be poised above address **$3F03**, ready to assemble the instruction LDY immediate #$0480. Find mnemonic LD on page 30, and follow in the second column until you find LDY. This is one of a limited number of two-byte opcodes, and its hex representation is **$10 8E**. The 6809 is a newcomer, based on the 6800 microprocessor. Opcodes like LDY are additions to the original 6800 instructions; where there's no room to fit an opcode in the binary instruction set, certain bytes are set aside as doorways into further instructions. The hex codes **$10** and **$11** serve that purpose; later on, check page 29 for a list of these.



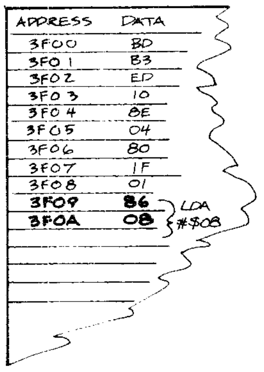
ADDRESS	DATA	
3F00	BD	
3F01	B3	
3F02	ED	
3F03	10	LDY
3F04	8E	
3F05	04	#$0480
3F06	80	

Back to the program. The opcode for LDY, then, is **$10 8E**. So across from address **$3F03**, write **$10, and across from address $3F04**, write **$8E**. Since this is an immediate instruction, the next two bytes are the operand. Next to addresses **$3F05** and **$3F06**, write the bytes **$04** and **$80**, respectively. You have now assembled the second program command.



ADDRESS	DATA	
3F00	BD	
3F01	B3	
3F02	ED	
3F03	10	
3F04	8E	
3F05	04	
3F06	80	
3F07	1F	TFR
3F08	01	D,X

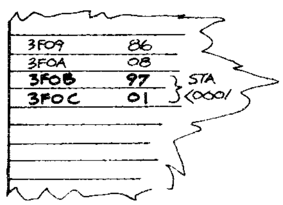
Those two were easy. The next instruction is **TFR D,X** (transfer D to X), which you can find on page 31. You'll find this in the immediate column, although that's stretching the point. The opcode is **$1F**, so write that next to address **$3F07**. The operand is D,X. Turn to page 34, where you'll find a block labeled "Transfer/Exchange Post Byte". This byte is divided into two four-bit blocks, that is, into two nybbles. The left-hand nybble is the source register, and the right-hand nybble is the destination register. The binary information below names the registers. Your program is transferring D to X. The source register is D, the destination register is X. Checking the table, you find that D is value **0000** and X is value **0001**. The combined byte is therefore **0000 0001**, or hex **$01**. Across from memory location **$3F08**, write **$01**. The opcode and operand for **TFR D,X** assemble to **$1F 01**.



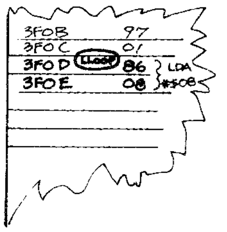
ADDRESS	DATA	
3F00	BD	
3F01	B3	
3F02	ED	
3F03	10	
3F04	8E	
3F05	04	
3F06	80	
3F07	1F	
3F08	01	
3F09	86	LDA
3F0A	08	#$08

Next. LDA immediate with 8. Back on page 30, under the LD instruction, you can find LDA. Since this is an immediate instruction, the opcode is **$86**. Next to address **$3F09**, write **$86**. The instruction is immediate, so the data is 8. Write **$08** across from address **$3F0A**. Things are moving now.

The instruction is STA Direct Page <0001. STA is found on page 31 under the instruction ST. This is a direct



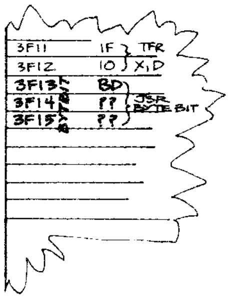
addressing mode, so the operand under the direct heading is **$97**. Write **$97** across from address **$3F0B**. In a direct instruction, the page is known, so only the least-significant byte is used as the operand. The address is **$0001** on page **$00**, so the least-significant byte is **$01**. That's the operand; write **$01** next to address **$3F0C**.



The next two instructions are virtually identical. LDA immediate 8 is again **$86 08**. Write **$86** next to **$3F0D**, and **$08** next to **$3F0E**. STA Direct Page <0000 is also very similar, assembling to **$97 00**. Write **$97** next to **$3F0F**, and write **$00** next to **$3F10**. The only thing to keep in mind is the label LLOOP, an abbreviation for Line Loop. Your program needs to come back to that address **$3F0D** each time it has to display a new line, so mark that label down on the bottom of the last page of your papers: write LLOOP, and across from it write the address **$3F0D**.

You're only 16 bytes into the program. I've already told you it will run nearly 100 bytes, so you're probably beginning to conclude that this assembly language stuff isn't for you. Hang on! The editor/assembler will do this all for you in seconds, but I'm convinced it won't do you any good to assemble everything by machine. There are two advantages to hand assembly: first, by the time you've hand assembled a program, you know it intimately. Second, if you're ever in a bind and need a quick diagnostic program, POKEing values into place may be the only solution. You have to be able to assemble a program from the data booklet, or you're wasting your time learning about this powerful 6809 processor.

Back to work. Transfer X to D — **TFR X,D**. The opcode you've used. Next to address **$3F11** write **$1F**, the transfer opcode. This time the source register is X and the destination register is D. If you've forgotten, turn to page 34. X register is binary **0001**, D register is binary **0000**. The composite byte made from these two nybbles is **0001 0000**, or hexadecimal **$10**. That's the operand. Next to address **$3F12**, write **$10**.



The next instruction is **JSR BYTBIT**. You've used the opcode for Jump to Subroutine (JSR) — that's **$BD**. Write **$BD** next to address **$3F13**. But how do you deal with the operand? You know it's an extended operand, which means it's two bytes. The subroutine BYTBIT is within the program you're writing, but you don't know its address yet. What you do now is leave two blank spaces at addresses **$3F14** and **$3F15**. You'll fill them in later when you know what they are. There are two pass-throughs to any assembly process, and this is the first pass.

The next free address is **$3F16**. The command is transfer, **$1F**. Write that next to **$3F16**. The transfer is from B to A. Again, turn to page 34. The source register is B, binary nybble **1001**; the destination register is A, binary nybble **1000**. The combined byte is **1001 1000**, or hex **$98**. Next to address **$3F17**, write **$98**.

```
* Hand assemble CMPB $FFFF

$F1 FF FF

* Hand assemble EORB $0001

$F8 00 01

* Hand assemble JMP $B3ED

$7E B3 ED

* Hand assemble LDX $7FFF

$BE 7F FF

* Hand assemble LDY $7FFF

$10 BE 7F FF

* Hand assemble LSR $0100

$74 01 00

* Hand assemble STD $0DDC

$FD 0D DC

* The following inherent
instructions do not appear in
the program. Hand assemble each
one.

* Hand assemble ASRA

$47

* Hand assemble CLRB

$5F

* Hand assemble COMA

$43

* Hand assemble INCB

$5C

* Hand assemble LSLB

$58

* Hand assemble NEGA

$40

* Hand assemble RORA

$46

* Hand assemble RTS

$39

* The following register
instructions do not appear in
the program. Hand assemble each
one.

* Hand assemble PULS A,CC,X,Y

$35 35
```

```
* Hand assemble PSHS
A,B,X,Y,CC,U,DP,PC

$36 FF

* Hand assemble TFR DP,B

$1F B9

* The following indexed
instructions do not appear in
the program.  Hand assemble each
one.

* Hand assemble CMPA ,Y

$A1 A4

* Hand assemble CMPA ,Y+

$A1 A0

* Hand assemble CMPA 5,Y

$A1 25

* Hand assemble CMPA $7F,Y

$A1 A8 7F

* Hand assemble CMPA $1234,Y

$A1 A9 12 34

* What does CMPA ,Y+ mean?

Compare A accumulator to memory
indexed by the Y register, with
no offset, and automatically
increment Y.

* What is hand assembly?

Figuring the hex (binary) code
byte by byte from the mnemonic
(source) code.
```

Another JSR to BYTBIT is next. Write the opcode for JSR, hex **$BD**, next to address **$3F18**, and leave blank spaces at **$3F19** and **$3F1A**. Again, when you find out where the subroutine BYTBIT is, you'll fill those in.

A LDA immediate is next. That instruction's been used before; the opcode is **$86**, the operand here is an immediate value, **$2A**. Write **$86** and **2A** next to addresses **$3F1B** and **$3F1C**, respectively.

LDB is a similar opcode to LDA. You'll find it right below; LDB immediate is **$C6**. Write **$C6** next to address **$3F1D**, and write its immediate operand, **$20**, next to address **$3F1E**.

On to **STB** ,Y+. Find the ST instructioon on page 31, and locate STB in the indexed addressing mode. The opcode is **$E7**. Next to address **$3F1F**, write **$E7**. In the column labeled "number of bytes", it says "2+", meaning this instruction requires a total of 2 or more bytes to complete. You have to determine how many and what they mean. Hand-assembling indexed addressing is the trickiest, but zero-offset indexed isn't bad. That's what you have here.

Turn to page 33. Find the table entitled "Indexed Addressing Postbyte Register Bit Assignments". This one byte contains a bucketful of information. It identifies the register, what kind of addressing mode is used with that register, and whether the addressing is non-indirect or indirect. I haven't talked about indirect addressing, so don't worry about that yet. In the right-hand column of this table is a description of each addressing mode; "EA" means effective address, that is, the address the instruction will calculate and use. The mode used in this instruction is auto-increment, zero-offset. That's the second mode down. The definition of "RR" is shown below the table. Your instruction uses the Y register, so RR is 01. Plug 01 into the binary digits shown, and the resulting number is **10100000**. The postbyte for the Y register in zero-offset indexed, auto-increment mode is hex **$A0**. There's your operand. Next to address **$3F20**, write **$A0**.

Between now and the next session, use your MC6809E data booklet to complete the rest of the program. If the process is still unclear, review the session up to this point. Don't cheat on me, now. When you can do this hand assembly without your hand held by me, then you're ready to go on. Talk to you then.