12.

Hello again. I hope you have been successful in your hand assembly of the remainder of the program. Here's a summary of what you should have been doing. . .

The next three instructions are easy. STA indexed is **$A7**. Write **$A7** next to address **$3F21**. The operand is zero-offset indexed; auto-increment Y register is the same as before. Across from address **$3F22**, write **$A0**. The following instruction is the same, **$A7 A0**. Write **$A7 A0** next to **$3F23** and **$3F24**, respectively. Finally, **STB ,Y+** comes around again. You know that's **$E7 A0**, so write **$E7 A0** next to **$3F25** and **$3F26** in turn.

Since you can use the table on page 33, the next instruction should strike no fear. It's **LDA ,X+**. Load A indexed, from page 30, is **$A6**. Write **$A6** next to address **$3F27**. Now glance at the chart on page 33. This is still auto-increment indexed, which is the second line of the table. The register is X, meaning the value for "RR" is 00. Plug 00 into the blank, and the binary byte becomes **1000 0000**. That's hex **$80**, and that's your operand. Next to address **$3F28**, write **$80**. And be sure to note the label BLOOP here at address **$3F27**. You've got to get back there later.

There's nothing really new in the rest of the main program, just tedious hand assembly. The next instruction is a **JSR**. That's hex code **$BD**. Write **$BD** next to address **$3F29**, and leave the next two addresses blank. Still don't know where the subroutine will be.

STB ,Y+ is next, and you can steal that information from earlier. STB indexed is **$E7**; write that at address **$3F2C**. Auto-increment zero-offset indexed Y is **$A0**; write that at address **$3F2D**.

The decrement instruction is next. Find that on page 30. This is decrement a direct page memory location you're dealing with, opcode **$0A**. Next to **$3F2E** write **$0A**. The location to decrement is **$00**, so that's your operand. Write **$00** next to address **$3F2F**.

Hand assembly is tiresome and troublesome. But it teaches you, giving you a level of intimacy with the machine that you can't achieve with mnemonics alone. If this kind of detail bothers you, consider that understanding someone else's program -- without recourse to commented source code -- can only be achieved by disassembling and examining the binary information. Knowing it both ways is your key to programming versatility.

* What is hand assembly?

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

* What are the bytes in an indexed instruction?

The opcode, the postbyte, and additional bytes of operand if necessary.

* Hand assemble LDA ,X+

$A6 80

* What is LDA ,X+ in binary?

10100110 10000000

* Hand assemble LDA $1234,X

$A6 89 12 34

Conditional branches

* What indexed addressing mode is LDA $1234,X?

16-bit constant-offset indexed.

* If the label BLOOP is found at address $3F27, hand assemble this instruction, found at address $3F30: BNE BLOOP

$26 F5

* What addressing mode is this?

Relative addressing.

* Relative addressing is relative to what?

The program counter (PC).

* In the assembly of BNE BLOOP ($26 F5), what does the value $F5 signify?

An offset relative to the program counter.

* What is the offset in binary?

In binary, 11110101

* What is the offset in decimal?

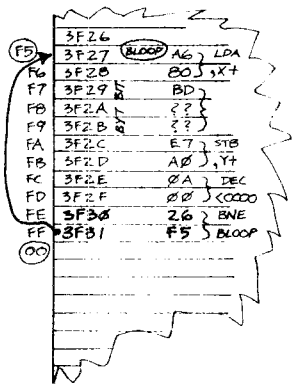
In decimal, -11.

* What makes $F5 negative?

The fact that in $F5 (11110101), the leftmost bit is a one.

* The following exercises are hand disassembly, that is, the translation from hexadecimal (or binary) code into mnemonic code. This is done with unknown programs for purposes of examining the operation of the program. Use the chart in the MC6809E data booklet on pages 28 and 29 for help. Disassemble, describe and give the mnemonic for each of the following groups of bytes.

Finally a branch instruction. Branch on Not Equal can be found on page 32. Find the table at the bottom right labeled "Simple Conditional Branches". Under "false", second mnemonic down, is BNE. The opcode shown is **$26**. So next to address **$3F30**, write **$26**. At this point in the program, the Program Counter is pointing to the next instruction in line after this one . . . meaning the Program Counter is pointing to address **$3F32**. Now locate your label BLOOP. This is where the branch is going. If **$3F32** is relative position **00**, count backwards to the address BLOOP, which is **$3F27**. **FF, FE; FD, FC; FB, FA; F9, F8, F7; F6, F5. $F5** is the position of BLOOP relative to the Program Counter. That makes **$F5** your operand for the relative branch BNE. Next to address **$3F31**, then, write **$F5**.

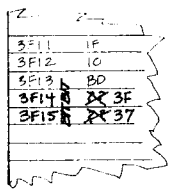


Decrement direct page you know already. The opcode is **$0A**, and should be written next to address **$3F32**. The operand for direct page **$00**, least significant byte **$01**, is **$01**. Next to address **$3F33**, write **$01**.

Another relative branch follows. This is BNE again, opcode **$26**. Write that down next to address **$3F34**. Now comes the counting backwards from the Program Counter, which is pointing to **$3F36**. You've got to get all the way back to LLOOP at address **$3F0D**. If you subtract it instead of counting backwards, you'll get the value **$D7**. I won't put you through it this time. Just write your relative branch operand **$D7** at address **$3F35**.

All that remains of the main program now is the return from subroutine. Find that on page 31 if you need to. It's opcode **$39**. Next to address **$3F36**, write **$39**. The main program is complete. Only the subroutines remain; the subroutine BYTBIT is coming up next, and its address is **$3F37**.

The subroutine BYTBIT begins at **$3F37**, meaning your three blank operands earlier in the program were filled with that address. The first action of the subroutine was to push the A Accumulator on the stack. **$34** is the opcode, and using the push/pull order chart, you found that **$02** is the operand. Four logical shift right A accumulator commands followed; each of these is **$44**.



Two more subroutine calls follow, **$BD** being the opcode for jump to subroutine. The addresses, which you had to calculate on your second assembly pass, are respectively **$3F4E** and **$3F58**.

Pull accumulator is **$35 02**, the operand calculated in the same manner as for the push command. And A immediate with **$0F** is represented **$84 0F**.

A familiar pair of subroutine calls follows — **$BD 3F 4E** and **$BD 3F 58** — and the convert and display subroutine finishes with the return from subroutine, **$39**.

The short CONVRT subroutine compares A immediate with **$0A** — that's **$81 0A**. It branches on carry clear (or

BHS . . . branch on high or same, meaning greater or equal) to the label LETTER. You calculated that relative branch to be **$03**, giving an instruction of **$24 03**. Add A immediate with **$30** is **$8B 30**, and return from subroutine is again **$39**.

At the label LETTER, add A immediate with **$37** is **$8B 37**, followed by **RTS**, **$39**.

Finally, the short display and update routine is made up of **STA ,Y+** . . . store A at Y, zero-offset, auto-increment. That pattern is familiar enough to copy the information from earlier in the program — **$A7 A0**. And, at last, the final return from subroutine, **$39**.

Your program should run from address **$3F00** to **$3F5A**, a total of 91 bytes. Look in your documentation, and see if your hand-assembled hexadecimal code agrees with mine:

```
3FØØ ** BD B3 ED 1Ø 8E Ø4 8Ø 1F
3FØ8 ** Ø1 86 Ø8 97 Ø1 86 Ø8 97
3F1Ø ** ØØ 1F 1Ø BD 3F 37 1F 98
3F18 ** BD 3F 37 86 2A C6 2Ø E7
3F2Ø ** AØ A7 AØ A7 AØ E7 AØ A6
3F28 ** 8Ø BD 3F 37 E7 AØ ØA ØØ
3F3Ø ** 26 F5 ØA Ø1 26 D7 39 34
3F38 ** Ø2 44 44 44 44 BD 3F 4E
3F4Ø ** BD 3F 58 35 Ø2 84 ØF BD
3F48 ** 3F 4E BD 3F 58 39 81 ØA
3F5Ø ** 24 Ø3 8B 3Ø 39 8B 37 39
3F58 ** A7 AØ 39
```

Time to get it running. I've got this batch of hexadecimal code prepared for you as a series of BASIC DATA statements.

Program #18, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs, rewind to the start of the program and try again. For severe loading problems, see the Appendix.

```
10 DATA BD,B3,ED,1Ø,8E,Ø4,8Ø,1F
15 DATA Ø1,86,Ø8,97,Ø1,86,Ø8,97
2Ø DATA ØØ,1F,1Ø,BD,3F,37,1F,98
25 DATA BD,3F,37,86,2A,C6,2Ø,E7
3Ø DATA AØ,A7,AØ,A7,AØ,E7,AØ,A6
35 DATA 8Ø,BD,3F,37,E7,AØ,ØA,ØØ
4Ø DATA 26,F5,ØA,Ø1,26,D7,39,34
45 DATA Ø2,44,44,44,44,BD,3F,4E
5Ø DATA BD,3F,58,35,Ø2,84,ØF,BD
55 DATA 3F,4E,BD,3F,58,39,81,ØA
6Ø DATA 24,Ø3,8B,3Ø,39,8B,37,39
65 DATA A7,AØ,39
7Ø FORX=&H3FØØ TO &H3F5A
75 READA$:A=VAL("&H"+A$)
8Ø POKEX,A:NEXT:DEFUSRØ=&H3FØØ
85 CLS:PRINT"TEST ADDRESS 3FØØ:"
9Ø M=USRØ(&H3FØØ)
```

*** Disassemble $BD B3 ED**

$BD is jump to subroutine, extended addressing mode; therefore, $BD B3 ED is JSR $B3ED.

*** Disassemble $86 6A**

$86 is load A accumulator immediate; therefore, $86 6A is LDA #$6A.

*** Disassemble $44**

$44 is an inherent instruction, logical shift right A accumulator: LSRA.

*** Disassemble $35 02**

$35 is pull from the hardware stack; $02 is binary 00000010, indicating the A accumulator. Therefore, the instruction is PULS A.

*** Disassemble $1F 01**

$1F is transfer from register to register; 01 is binary 0000 0001. The transfer-from register is D (0000) and the transfer-to register is X (0001). Therefore the instruction is TFR D,X.

*** Disassemble $10 8E 04 80**

$10 8E is a two-byte opcode for load Y register immediate; the Y register is 16 bits, so $04 80 is the 16-bit operand. Therefore, the instruction is LDY #$0480.

*** Disassemble $81 0A**

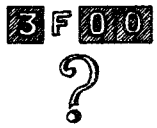
$81 is compare A register immediate; therefore the instruction is CMPA #$0A.

*** Disassemble $0A 01**

$0A is the opcode for decrement memory direct; therefore the instruction is DEC ($NN01, where NN is the direct page register.

```
* Disassemble $86 6A C6 60

$86 is load A accumulator
immediate, so $86 6A is LDA
#$6A. That means $C6 60 must be
another instruction.  $C6 is
load B accumulator immediate, so
$C6 60 is LDB #$60.  You can't
fool me.

* Disassemble $A7 A0

$A7 is store A accumulator
indexed; A0 is binary 10100000.
Referring to the chart, the only
postbyte that ends in 0000 is
,R+.  1RR0 applied to 1010 makes
RR=01.  01 is the Y register.
Therefore, $A7 A0 is STA ,Y+.

* Disassemble $A6 80

$A6 is load A accumulator
indexed.  $80 is 10000000.  This
is again ,R+ indexed mode, with
1RR0 = 1000, RR = 00 = X
register.  Therefore, $A6 80 is
LDA ,X+.

* What does &H mean in BASIC?

Hexadecimal.

* If A$="BD", what is the value
of A after this statement:
A=VAL("&H"+A$)?

A equals decimal 189 (hex $BD).

* What does DEFUSR0=&H3F00
mean?

Define the BASIC user entry
point number 0 to be at $3F00.

* How many characters (letters,
numbers and symbols) does the
Color Computer display using
PRINT?

96.

* How many characters is the
Color Computer capable of
displaying using POKE?

128.
```

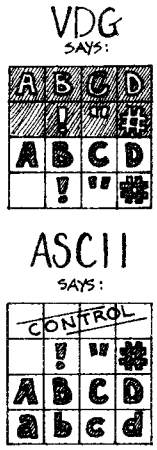
So there it is. A reasonably painful first hand-assembly, resolved into a mere 12 lines of BASIC DATA statements, POKEd in place and used as a subroutine via the USR command. In this test program, the address transferred for display is **$3F00** — so you can look at the machine language program itself. Will it work? No, it won't. That is, not exactly as you expect. RUN the program.



Well, you *are* looking at your own hand-assembly, but something's amiss. The letters are okay, but the numbers are shown in reverse video.

A peculiarity like this is one of my reasons for preparing these lessons with the Color Computer in mind. If you've been using your Color Computer for a while, you know that upper case characters, plus numbers and symbols, are presented normally, but that lowercase characters are represented by reverse video. What you're running into here is the video display generator, the VDG. There's a software shuffle done by BASIC to accept your ASCII information and translate it into VDG codes.

It's bit time again. The video display generator contains only 64 letters, numbers and symbols, all standard uppercase characters. No lowercase or control characters were included in the design and manufacture of this part. To display any character in this set, then, only bits 0 through 5 in a byte are used. However, bits 6 and 7 are connected to the VDG. Bit 7 turns on the low-resolution color graphics, which BASIC calls CHR$(128) through CHR$(255) — hexadecimal **$80** through **$FF**. Bit 6 is the tricky one. It is used to turn on the inverse-video mode for the alphanumeric characters. When bit 6 is a one, normal characters are seen; when bit 6 is a zero, reverse characters are displayed.

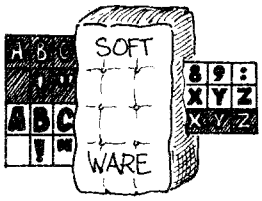


Think back to my example of POKEing vs. PRINTing the screen, way back in the first session. PRINTing the characters resulted in their appearance in normal ASCII order — control characters from **$00** to **$1F** were not displayed, **$20** through **$3F** were numbers and symbols, **$40** through **$5F** was uppercase, and **$60** through **$7F** was reverse-video-style lowercase.

But POKEing the values to the screen resulted in something different. Values **$00** through **$1F** revealed reverse-video-style lowercase, **$20** through **$3F** displayed a not-before-seen group of reverse-video numbers and symbols, **$40** through **$5F** showed the uppercase characters in their proper ASCII position, and **$60** through **$7F** displayed the normal set of numbers and symbols.

The reasons should begin to come clear. If bit 7 is zero, then alphanumerics are displayed instead of graphics. If bit 6 is zero, all characters are displayed in reverse video mode. In other words, the hardware of the Color Computer understands that all characters from **00 00 0000** to **00 11 1111** — that is, from hex **$00** to **$3F** — are reverse characters. Conversely, if bit 6 is one, all characters are

displayed in normal video mode. The Color Computer hardware then understands that all characters from **01 00 0000** to **01 11 1111** — that is, from hex **$40** to **$7F** — are normal characters.



The BASIC language works with ASCII, so this hardware business is a pain. BASIC is forced to translate ASCII to hardware and hardware to ASCII every time it does a screen display! So whenever you write software in machine language, you will also have to provide some sort of translation. Here's a summary:

If you want:	You have to use:
ASCII $00 to $1F, control functions	Control software without display.
ASCII $20 to $3F, numbers and symbols	Hardware $60 to $7F
ASCII $40 to $5F, normal uppercase	Hardware $40 to $5F (no change).
ASCII $60 to $7F, normal lowercase	Hardware $00 to $1F

In normal display (such as BASIC), hardware values **$20** to **$3F** are not used; these are the reverse numbers and symbols. The program you just created, in attempting to use legitimate ASCII values, used the hardware values for reverse characters. That accounts for the funky screen display.

$2A = *
+ $40
$6A = *

$2A = 0010 1010
$6A = 0110 1010

Now you have enough information to get out of that dilemma. Turn back to your hand-assembled listing and locate the spots where a display character is established. You'll find address **$3F1C** is supposed to be a star, hex **$2A**. Glance at your documentation where the summary I just gave you is printed. If you want to display **$2A**, then, you actually need to use the hardware value **$6A**. Put that in place. Type POKE &H3F1C,&H6A and hit <ENTER>. That's POKE &H3F1C,&H6A <ENTER>. That should give you a proper star; try it. Type GOTO85 and hit <ENTER>.

The stars are okay now. The spaces are next. A space is **$20**, which means the hardware requires a **$60**. In your hand assembly, you'll find that space at address **$3F1E**. Change it now. Type POKE &H3F1E,&H60 and hit <ENTER>. That's POKE &H3F1E,&H60 <ENTER>. The spaces should be cleared up. Type GOTO85 and hit <ENTER>.

$32 = 2
+ $40
$72 = 2

$32 = 0011 0010
$72 = 0111 0010

Only the reverse numbers remain to cure. This happened in the ASCII conversion subroutine that began at address **$3F4E**. Find that subroutine. At address **$3F52**, an offset of **$30** was added to convert from the number 0 through 9 to ASCII CHARACTERS "0" through "9". Hex values for these are **$30** through **$39**, meaning the hardware needs **$70** through **$79** to present the numbers correctly. So the

* What does VDG mean?

Video display generator.

* How many unique characters is the VDG capable of displaying?

64.

* Why can the VDG display 64 characters, whereas the Color Computer can display 128?

Because the Color Computer displays 64 normal characters and 64 reverse-video characters.

* What do the ASCII codes from $00 to $1F represent?

Control codes (carriage return, backspace, tab, etc.)

* What do the ASCII codes from $20 to $3F represent?

Numbers, symbols and punctuation.

* What do the ASCII codes from $40 to $5F represent?

Uppercase (capital) letters.

* What do the ASCII codes from $60 to $7F represent?

Lowercase (small) letters.

* What do VDG codes $00 to $1F represent?

Lowercase (reverse) letters.

* What do VDG codes $20 to $3F represent?

Reverse-video numbers, symbols and punctuation.

* What do VDG codes $40 to $5F represent?

Uppercase letters.

Program #19

*** What do VDG codes $60 to $7F represent?**

Numbers, symbols and punctuation.

*** To create the display "A0D7 ** 8E" in ASCII, what ten bytes would be used?**

$41 30 44 37 20 2A 2A 20 38 45

*** To create the display "A0D7 ** 8E" in VDG terms, what ten bytes would be used?**

$41 70 44 77 60 6A 6A 60 78 45

offset at address **$3F53** has to be changed from a proper ASCII **$30** to the hardware's demand of **$70**. Do it. POKE &H3F53,&H70 and hit <ENTER>. That's POKE &H3F53,&H70 <ENTER>. That should cure the numbers. Type GOTO85 and hit <ENTER>.

That did it. The address and data display is complete. That video hardware shuffle is a little tricky, so if it's not clear to you at this point, please review from the start of this session. You can break now. Otherwise, I have a program for you to load.

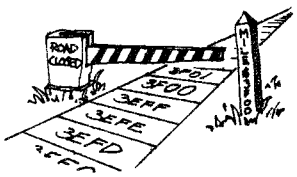
Program #19, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, type L and press ENTER. The computer will search (S) and find (F). When the cursor reappears, display the program. Type P#:* and press ENTER. If the right-hand side of the program 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.

```
            B3ED      00100 INTCNV  EQU     $B3ED
            0000      00110 BYTES   EQU     $0000
            0001      00120 LINES   EQU     $0001
                      00130 *
3F00                  00140         ORG     $3F00
                      00150 *
3F00 BD     B3ED      00160         JSR     INTCNV
3F03 108E   0480      00170         LDY     #$0480
3F07 1F     01        00180         TFR     D,X
3F09 86     08        00190         LDA     #8
3F0B 97     01        00200         STA     <LINES
3F0D 86     08        00210 LLOOP   LDA     #8
3F0F 97     00        00220         STA     <BYTES
3F11 1F     10        00230         TFR     X,D
3F13 BD     3F37      00240         JSR     BYTBIT
3F16 1F     98        00250         TFR     B,A
3F18 BD     3F37      00260         JSR     BYTBIT
3F1B 86     6A        00270         LDA     #$6A
3F1D C6     60        00280         LDB     #$60
3F1F E7     A0        00290         STB     ,Y+
3F21 A7     A0        00300         STA     ,Y+
3F23 A7     A0        00310         STA     ,Y+
3F25 E7     A0        00320         STB     ,Y+
3F27 A6     80        00330 BLOOP   LDA     ,X+
3F29 BD     3F37      00340         JSR     BYTBIT
3F2C E7     A0        00350         STB     ,Y+
3F2E 0A     00        00360         DEC     <BYTES
3F30 26     F5        00370         BNE     BLOOP
3F32 0A     01        00380         DEC     <LINES
3F34 26     D7        00390         BNE     LLOOP
3F36 39               00400         RTS
                      00410 *
3F37 34     02        00420 BYTBIT  PSHS    A
3F39 44               00430         LSRA
3F3A 44               00440         LSRA
3F3B 44               00450         LSRA
3F3C 44               00460         LSRA
3F3D BD     3F4E      00470         JSR     CONVRT
3F40 BD     3F58      00480         JSR     DISPLY
3F43 35     02        00490         PULS    A
3F45 84     0F        00500         ANDA    #$0F
3F47 BD     3F4E      00510         JSR     CONVRT
3F4A BD     3F58      00520         JSR     DISPLY
3F4D 39               00530         RTS
```

```
                        00540 *
3F4E 81   0A            00550 CONVRT  CMPA     #$0A
3F50 24   03            00560         BCC      LETTER
3F52 8B   70            00570         ADDA     #$70
3F54 39                 00580         RTS
3F55 8B   37            00590 LETTER  ADDA     #$37
3F57 39                 00600         RTS
3F58 A7   A0            00620 DISPLY  STA      ,Y+
3F5A 39                 00630         RTS
          0000          00640         END
00000 TOTAL ERRORS
BLOOP   3F27
BYTBIT  3F37
BYTES   0000
CONVRT  3F4E
DISPLY  3F58
INTCNV  B3ED
LETTER  3F55
LINES   0001
LLOOP   3F0D
```

Here's the complete program you just created. You have the entire mnemonic listing available, which the assembler can convert to machine language very quickly.

You'll assemble this, go right into BASIC, and load the next program on the tape. Here's how it goes. Type A/IM/AO and hit <ENTER>. The listing will scroll by, and the program will be assembled at **$3F00**. When the star prompt and cursor return, quit the editor/assembler: Type Q <ENTER>. In a few seconds, the Extended Color BASIC message will appear. You know the program is at **$3F00**, so protect memory.



If you've never protected memory before, the purpose is to tell BASIC that a certain area is off-limits. BASIC will make no attempt to use protected memory, except through PEEK, POKE andDM statements. You can refer to your BASIC manual for details. Type CLEAR 200,&H3F00 and hit <ENTER>. That's CLEAR 200,&H3F00. Now you can load the next program.

Program #20, a BASIC program. Turn on the power of your Extended Color BASIC computer. When the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/0 error occurs, rewind to the start of the program and try again. For sever loading problems, see the Appendix.

```
10 DEFUSR0=&H3F00
20 CLS
30 PRINT@0,"":PRINT@0,"";
40 INPUT"ADDRESS";A$
50 A=VAL("&H"+A$)
60 IFA>32767THENA=A-65536
70 M=USR(A)
80 GOTO30
```

*** What is the process of translating mnemonic (source) code into binary (object) code called?**

Assembly.

*** What is the process of translating binary (object) code into mnemonic (source) code called?**

Disassembly.

*** What is the term for binary digit?**

Bit.

*** What is the term for eight binary digits (bits)?**

A byte.

*** What is the term for four binary digits?**

A nybble (also spelled nibble).

*** What number system represents binary digits?**

The binary system.

*** What number system organizes the binary numbers into convenient size?**

The hexadecimal system.

Summary: Architecture

*** What does ASCII mean?**

The American Standard Code for Information Interchange.

*** What are the ASCII codes?**

Control codes; numbers, symbols, and punctuation; and upper and lowercase letters.

*** What is the term for a computer instruction?**

An opcode.

*** What is the term for an opcode's data?**

An operand.

*** What is the term for the design and purpose of a processor?**

Its architecture.

*** What is the architecture of the 6809 processor?**

Accumulators A and B; index registers X and Y; stack pointers S and U; direct page register DP; condition code register CC; program counter PC; 65,536 bytes of memory.

*** What is an addressing mode?**

The way in which a machine language program gets its information.

*** What are the 6809's addressing modes?**

Inherent, register, immediate, extended, direct, and indexed.

If you LIST this program, you can see that it is simply a little BASIC input routine which subsequently calls your machine-language subroutine. Since the value transferred is an integer, the range is –32768 to +32767. To get at addresses higher than 32767 (that's hex **$7FFF**), there has to be a conversion. Line 60 is a little trick that's good to know. So assuming the assembly went well, then your program should reside at **$3F00** right now. RUN the program.

Enter any address from **$0000** to **$FFFF**. There's memory, 64 bytes of it. Play around with this program for a little bit, and then come back to this tape for a summary.

> Please examine a number of areas of memory using this BASIC program. When you are confident of the significance and application of this process, return to the tape.

This course is halfway now. If you're a newcomer to assembly language, you're probably just a little overwhelmed. The jargon and the concepts take *do* some time to settle. On the other hand, if you're an experienced programmer here just to learn the specifics of the 6809, then I know you're itching to get on with it. In either case, I would like you to stay with me for a summary of the main points from these past twelve sessions.

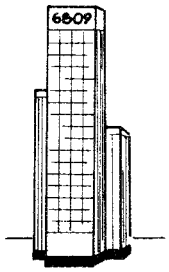
Assembly language is not BASIC, but forms a perfect companion to BASIC, as you have seen with the programs so far. It is capable of easy access to functions not easily available via BASIC, such as reverse numbers and symbols, and is fast and flexible. From assembly language is built the language of BASIC itself.

Assembly language represents computer instructions. Computer instructions are actually electronic signal patterns best represented by the binary number system. Binary numbers are difficult to recognize, so binary patterns are visually organized by using a single symbol for each group of four bits. This is the purpose of the hexadecimal numbering system **1** through **9** and **A** through **F**. Logical arrangements and patterns of binary digits were used in the creation of the American Standard Code for Information Interchange, ASCII.

A processor can interpret binary patterns as either instructions or data. The instruction pattern is known as the opcode, and the data pattern is called the operand. All binary patterns in a program are found in memory, and it is only the order and context which differentiate opcodes from operands.

Each processor has a unique design and purpose. This

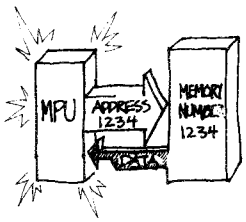
design and purpose is known as the processor's architecture. The architecture of the 6809 is particularly strong in the way it accesses information. The architecture of the 6809 consists of a program counter PC, two arithmetic-performing accumulators A and B, two index registers X and Y, two stack pointers S and U, a condition code register CC (also called the flags), and a direct page register DP.



To manipulate these registers through a program, the binary code must be presented to the instruction decoder. Because binary and hexadecimal representations are machine-level instructions with little value for the human programmer, instruction names are used to ease the programming process. They are called mnemonics. A program written in mnemonics cannot be executed; only the machine code the mnemonics represent can be executed. The process of translating mnemonics into machine code is called assembly. The program that permits editing mnemonic code, also called source code, is an editor. The program that translates this source code into machine code, also called object code, is an assembler. Usually these are combined in a single program known as an editor/assember; this series uses the program EDTASM+ as its editor/assembler.

A machine language program must access information. The way it finds this information in memory is called an addressing mode. There are several major addressing modes in the 6809 processor. Inherent addressing has the data implied as part of the opcode itself. Register addressing has the data available in one of the 6809 registers. Immediate addressing presents the data in the program memory immediately following the opcode. Extended addressing presents a complete 16-bit memory address at which the data can be found. Direct addressing presents an 8-bit address which is combined with the DP register to locate the data in memory.

Indexed addressing uses registers and offsets to calculate the address in memory at which data can be found. This mode is complex and flexible, with automatic incrementing and decrementing of registers as the instruction is executed. Relative addressing presents a value which directs the program to a position in memory relative to the current position of the program counter.



Information is accessed by processor instructions. Among these are store and load, which save and retrieve information in memory; arithmetic instructions such as add, subtract, decrement, increment, and negate; logical instructions such as AND, OR, complement, and Exclusive-OR; bit shift and rotation instructions; jumps and branches to other program locations or subroutines.

Programs and information are stored in memory. The organization of memory is called a memory map, which can contain read-write memory, read-only memory, special-purpose memory registers, and input/output ports.

Summary: Addressing

* What do RAM, ROM, CPU, SAM, PIA, and VDG mean?

RAM means read/write memory, also known as random-access memory; ROM means read-only memory; CPU means central processing unit; SAM means synchronous address multiplexer; PIA means peripheral interface adapter; and VDG means video display generator.

* What is the process of translating mnemonic (source) code into binary (object) code called?

Assembly.

* What programming tool performs this task?

An assembler.

* How many topics must you know to continue this course?

Six topics.

* What is the first topic you need to know to continue this course?

How to use the MC6809E data booklet.

* What is the second topic you need to know?

How to enter and edit programs using EDTASM+.

* What is the third topic you need to know?

How to count in binary and hexadecimal.

* What is the fourth necessary topic?

How to create BASIC programs which POKE machine language into memory.

Summary: Special devices

*** What is the fifth item you need to know?**

All the 6809 instructions presented up to the 12th session.

*** What is the final topic?**

The addressing modes presented up to the 12th session.

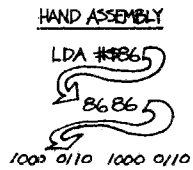
*** How many questions have you answered so far in this course?**

895. Bet you didn't know that.

The Color Computer has a specific memory map and several hardware devices. Read-write memory, or RAM, is located in the bottom two quarters of the memory map; the BASIC language in read-only-memory (or ROM) occupies the third quarter; most of the upper quarter is occupied by cartridge ROM when it is plugged in.

The top 256 bytes of memory have a special purpose, and are used by Peripheral Interface Adaptors (the PIAs) as input/output ports for the keyboard, cassette, printer, video display, and other reserved purposes. In the Color Computer, the most sophisticated of these functions is performed by the Synchronous Address Multiplexer (the SAM), which controls the memory circuitry, the processor speed, and the Video Display Generator (the VDG). By using a combination of the PIAs and the SAM, the VDG can be placed into several modes of alphanumerics and low- and high-resolution color graphics, and can be made to display any area of memory. Machine-language programs most easily control these devices.

Machine-language programs for control, display or any type of programming can be assembled using a tool such as the editor/assembler, or can be assembled by hand using a list of commands and their respective binary codes. Hand assembly is tedious, but is valuable for learning to create compact and efficient programs, and for understanding the specific actions taken by the processor. Confident hand-assembly can reveal peculiarities in a computer, such as the alphanumeric display method in the Color Computer.



So that's a very fast trip through the past twelve sessions. I recommend that you take a breather now and review these lessons, because I plan to pick up the pace from here on. Things you must know to continue are: how to use the MC6809E data booklet; how to enter and edit programs using EDTASM+; how to count in binary and hexadecimal; and how to create BASIC programs which POKE machine-language information into memory. You must also know the 6809 instructions that have been presented so far, and all the addressing modes which I've explained.

I won't have time to summarize all of this again, so if you think you need to, please review now. I can't emphasize enough the need to review, because I can tell you from experience that if you get in to this too deeply and your background is not secure, the new information will muddy the old information so badly it will all become useless.

Now that I've issued my dire warnings, I hope you will continue this series. I'll be presenting graphics and sound software soon, and giving you pointers on making your programs short, be quick, and run bug-free. Speak to you next time.

13.

Hello and welcome back to the final half of "Learning the 6809." The pace will quicken somewhat, so I hope you've given yourself a solid foundation in the essentials of machine language programming that I presented in the first half of this course.

The topic this session is timing: that's the careful organization of computer instructions to perform tasks at a known speed. Unlike mechanical timers or ordinary clocks, the computer operations you can be certain of actually simplify this task. You are certain of the clock speed, that is, the number of fixed pulses per second by which the processor completes its instructions. And, you can identify the specific number of those clock cycles each instruction requires, since this is consistent... and the full information is provided with the data booklet.

You may not be as impressed as I am with this concept. But consider that all the real-world interfacing of the computer depends on some sort of timing. Here are just a few of those interfacing tasks:

1. Communication with a printer is timed. A printer connected to the Color Computer expects precisely 600 binary digits per second.

2. Cassette input and output is astoundingly precise. Not only is the timing of the binary digits critical, but the shape of the sound's wave recorded on the tape is important. Care in these timings overcomes the inherently poor quality of portable cassette recorders.

3. Keyboard input even uses timing. As the metal contacts of the keys close, a little electromechanical bouncing takes place. This bounce must be timed through so as not to produce unwanted double or triple characters.

4. BASIC sound commands need frequency

The very speed that makes machine language a programming delight also makes it difficult when dealing with a real world operating in human terms. You start wishing for BASIC after a few hours of meticulously timed program actions. But you'll never be able to create sound or games with real punch and clarity from BASIC, so machine-language bit twiddling is the solution. Onward!

* What is the clock speed of the Color Computer?

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

* If a printer expects information at 600 binary digits per second, how many clock pulses is that?

Approximately 1,492 clock pulses.

* If a given computer activity had to take place 100 times per second, how many clock cycles would that be?

Approximately 89,489 clock cycles.

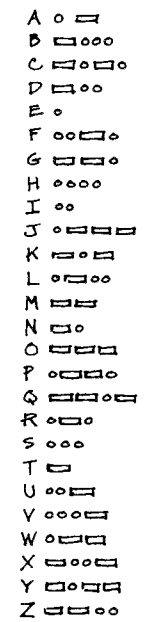
```
* At 1,000 activities per
second?

Approximately 8,949 clock
cycles.

* At 10,000 activities per
second?

Approximately 895 clock cycles.

* What does Hz mean?

Hz means Hertz, or cycles
(pulses) per second.

* What does MHz mean?

MHz means megaHertz, or million
cycles per second.

* Which of the following require
consideration of timing:
cassette input and output;
serial printer output; keyboard
input; sound output.

All require consideration of
timing.

* Why does cassette input and
output require timing?

Because the data must be
recorded and received at a known
rate.

* Why does serial printer output
require timing?

Because a serial printer must
receive data at a known rate.

* Why does keyboard input
require timing?

Because mechanical contact
bounce must be ignored (timed
through).

* Why does sound output require
timing?

Because sound is made up of
specific frequencies, and
frequencies are inherently
time-based.
```

information in order to produce proper musical pitches.

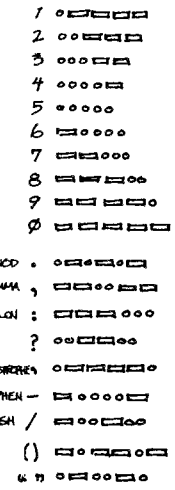
These four examples are only the most obvious. Subtle kinds of timing permeate machine language programming.

I'd like to start with the simplest kind of timing, the delay loop. No doubt you've used FOR-NEXT loops in BASIC to time such things as screen presentations and Inkey$ input. Another interesting use of delay loops is for simplified communications timing . . . in the example I've got for you, it's used for sending fast and accurate Morse Code. Now Morse Code might be a little bit of an anachronism in this computer era, but it's interesting and I think quite a lot of fun.

First, conceptualize the problem and establish some parameters. Morse code is that pattern of long and short beeps that has been used for over a century to communicate across telegraph wires and via radio. In this example, the code might be sent from the keyboard, or it might be sent from a prepared, edited message. Also, you've got to establish the speed of code transmission and choose the pitch of the beep. Finally, the character set to be used must be selected (that is, the whole set or just the alphabetic characters).



Let's take the last first, and say that the entire 6-bit ASCII character set should be used. Those are numbers and uppercase letters. Let's set the beep at a clear 1,000 Hz — 1,000 cycles or vibrations per second. And finally, establish the transmission speed at about 10 words per minute. Before actually programming these last two items, keep in mind that it might be wise to make both the beep frequency and the transmission speed flexible, so they can be changed by the operator to match the circumstances.



Now to the concept. It seems to break down into a few simple steps coupled with a some crucial subroutines. It looks like this. A message is found somewhere in memory. The code for each character is located in a table of Morse codes. After the code is identified, it is used in conjunction with two or three subroutines to produce beeps and silences of the proper timing.

Now I don't know very much about Morse Code, but from what I'm told, it consists of short and long beeps known as "dits" and "dahs." A "dah" is roughly three times as long as a "dit," and all beeps are separated by "dit"-length silences. Letters are separated by "dah"-length silences, and words, when separated at all, are separated by about two "dahs."

Before I get too far ahead, let me play for you a little bit of professional Morse Code . . .

What you just heard was the message "Hello how are you." It's a series of pure, regular beeping tones and silences. You might think that Extended Color BASIC has a perfectly adequate group of SOUND and PLAY commands, tailored to this kind of task. Unfortunately, they won't do for a number of reasons. First, the beep length is a fixed multiple of the shortest length. Morse Code speeds often fall in between these fixed lengths. Next, the BASIC programming is very clumsy, using a long array, substantive error-checking, and various loops. But worst of all is the slight but distinguishable "gargling" in the sound, an adulteration of the pure tone with pops and burbles. At first — and especially if you are listening on an inexpensive television — that impurity may be obscured by the limited TV sound. But if you listen through a separate amplifier hooked to the cassette output, the unevenness of the sound becomes distinct. Think about those things as you load and run the following BASIC program.

Program #21, a BASIC program. Turn on the power of your Extended Color BASIC computer. Whe the cursor appears, type CLOAD and press ENTER. The computer will search (S) and find (F). When the cursor reappears, LIST this program. If the program is not similar to the listing, or if an I/O error occurs. rewind to the start of the program and try again. For severe loading problems. see the Appendix.

```
1 CLEAR500:DIMA$(60)
2 FORX=32 TO 90
3 READA$(X-32)
4 NEXT
5 CLS
6 PRINT"TYPE YOUR MESSAGE"
7 PRINT"(MAXIMUM 255 CHARACTERS)"
8 LINEINPUTB$
9 PRINT:PRINT"SENDING MESSAGE..."
10 FORX=1TO LEN(B$)
11 A=ASC(MID$(B$,X,1))
12 A=A-32:PRINTMID$(B$,X,1);
13 C$=A$(A)
14 IFC$="SP"THEN22
15 PRINTC$" ";
16 FORY=1TOLEN(C$)
17 Q$=MID$(C$,Y,1)
18 IFQ$="."THENSOUND240,1
19 IFQ$="-"THENSOUND240,3
20 FORZ=1TO50:NEXT
21 NEXT
22 FORZ=1TO100:NEXT
23 NEXT
24 GOTO5
25 DATASP,SP,.-.-.,SP,SP,SP,SP,.----.,-.--.-,-.--.-,SP,SP,--..-
-,-....-,.-.-.-,-..-.
26 DATA------,.----,..---,...--,....-,.....,-....,--...,---..,--
---.,---...,SP,SP,SP,SP,...--..
27 DATASP,.-,-...,-.-.,-..,.,...-.,--.,.....,...,.---,-.-,.-..,--,-
.,---
28 DATA.--.,--.-,.-.,...,-,..-,...-,.--,-..-,-.--,--..
```

*** Is timing required for Morse Code?**

Yes.

*** Name the timing considerations needed for Morse Code.**

The length of the "dit", the length of the "dah", the length of silences, and the frequency of the beep.

*** How long is a "dah" with respect to a "dit"?**

Three times as long.

*** What commands produce sound in BASIC?**

SOUND and PLAY.

*** How long is the shortest BASIC beep (using SOUND X,1)?**

Approximately 1/14 of a second. (You weren't told that in the text).

*** If a loop contains two load A immediates, two store A extendeds, and one branch to make a complete loop, how many clock cycles are required for this loop?**

2 times 2 cycles, plus 2 times 5 cycles, plus 2 cycles ... a total of 16 cycles.

*** At 894,886 clock cycles per second, how many loops is this?**

894,886 divided by 16, or 55,930 cycles.

You've just heard a BASIC solution to the problem of transmitting Morse Code. For the simplest of purposes, this kind of code transmission might be adequate. But we can do far better in machine language. The timing and

* What is the main disadvantage of producing sound using BASIC?

The "gargling" or unevenness of the sound.

* What causes the "gargling" of the sound?

An interrupt.

* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* What is the process of acting on an interrupt called?

Servicing the interrupt.

* What causes an interrupt?

When an external signal line changes from one to zero.

* What three things happen when an interruot occurs?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* Can more than one interrupt occur?

Yes.

* Which interrupt gets taken care of?

The one with higher priority.

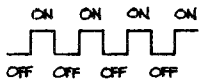
* What is the process of taking care of the interrupt called?

Servicing the interrupt.

control of the sound can be intimately precise, and that annoying gargle will disappear.

What, then, do you suppose causes that gargling sound? It seems to be a regularly recurring group of little hiccups as the tone proceeds. In fact, those hiccups are the time it takes the computer to briefly abandon the sound program in progress and perform other tasks. It is responding to an interrupt.

Practically all microprocessors are provided with electrical connections known as interrupt lines. When these interrupt lines are made to change from one to zero by some external happening, the microprocessor finishes its current instruction, saves important information, and follows special programming instructions in response to that interrupt.



It's like one of those drive-up fast-food places. We take you now to Sam's Roadside Kitchen in Roadside, New Jersey, where the sign reads "Honk for Sam's roadside drive-up service noon to 6 only. Other times honk at your own risk"..

Marge the Waitress: <indoors, talking to cook> One fries, two BLTs, three chili dogs . . . <honk> Alright, alright. <back to cook> . . . and one onion rings. Get those ready. There's a guy out honkin' that thing like Little Richard. <going outdoors> Yeah, what'll you have?

Car one: Three burgers, two fries, a shake.

Marge: Ya want bunny burgers or buddy burgers?

Car one: One bunny burger, two buddy burgers.

Marge: <indoors again>. One bunny, two buddies, fries. Where's my order? <to counter> Anything else, Joe? How 'bout you, Mac?

Mac: Yeah, gimme another dog, will ya Marge? With onions an' cheese, too.

Marge: <to cook> Cheese dog onions.

Kitchen: Orders up.

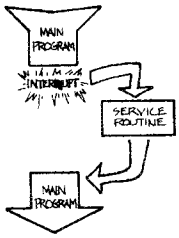
Marge: <to cook> Hey where's my steak? And what about. . . <honking> . . . the chili dog. Damn. Gotta get that. <outside again, honking continues> Yeah, yeah, whaddaya want?

Car two: Gimme three bunnies and... <honking from third car>

Marge: <to third car> Hey fella I'm busy. Sit on it till I get to ya. <back to car> Three bunnies. What else, and make it quick.

Car 2: How about filet mignon and truffles and leeks vinaigrette. . .

In this example, the restaurant was the computer, and Marge its microprocessor. The cook and customers served as program and storage memory. The car horn was the interrupt. Recall how Marge finished only the immediate task, and then went out to take care of the drive-up customer. In computer terms, that process is called "servicing the interrupt." When servicing an interrupt, the computer saves program counter and registers so that all information is intact when it returns from the interrupt service routine to finish its previous task. Interestingly, Marge chose to put the third honking customer on hold while she finished with the second. In that case, the interrupt in progress had a higher priority. Had the third car been an old favorite customer, however, Marge might have serviced that interrupt first, leaving the current task incomplete until the interrupt service routine was done.



Finally, Sam's sign said that drive-up service was from noon to six only. Other times, honking customers would be ignored. That process is known as masking an interrupt. They'll be much more talk about interrupts and how they are used later; right now, we only want to know how to get them out of the Morse Code beep. To do that, you have a little reading ahead.

> Please read the information on interrupts in the MC6809E data booklet. The condition codes are described at the top of page 6, the vectors are shown in Table 1 on page 6, and an explanation is presented in the first column on page 9. Return to the tape when you have completed the reading.

The most important thing you've discovered about interrupts, at least with respect to the Morse Code program, is that the interrupts can be turned off — masked, that is — by using the condition code register. Bits 4 and 6 are responsible for interrupts; there must be some way to use logical functions AND and OR with the condition code register to mask bits 4 and 6 in or out at will. There is.

Turn to the MC6809E data booklet, pages 30 and 31. You'll find that ANDCC is a special-purpose instruction available only in the immediate addressing mode; so is ORCC on the next page. You might want to check me on paper for what follows. If you wish to set bits 4 and 6 to one — that is, turn the interrupts off — you would OR the byte with binary **0101 0000**. Bits 0 through 3, 5 and would remain unaffected. To turn the interrupts on you need to set bits 4 and 6 to zero; to do that, you would AND the condition code register byte with **10101111**. In either case, the original condition codes for carry, negative, zero, half-

```
* Can interrupts be ignored?

Yes.

* What permits the processor to
ignore an interrupt?

Masking the interrupt.

* Is there an interrupt taking
place when BASIC is running?

Yes.

* What is one effect of the
interrupt?

A "gargling" in the SOUND
command.

* What causes the gargling?

The time taken to service the
interrupt; the interrupt service
routine.

* Can sound be produced without
"gargling"?

Yes.

* How can sound be produced
without gargling?

By producing it in machine
language.

* How can machine language stop
the gargling?

By turning off the interrupt.

* What is the proper term for
turning off an interrupt?

Masking the interrupt.

* What determines whether an
interrupt is masked or enabled?

The condition code register.

* What part of the condition
code register determines whether
an interrupt is masked or
enabled?

Bits 4 and 6.
```

* What masks an interrupt?

Setting its condition code bit to a one.

* What commands can be used to affect the condition code register directly?

ANDCC and ORCC, both immediate instructions.

* What condition code bits determine whether interrupts are masked or enabled?

Bits 4 and 6.

* What command specifically masks out (turns off) both interrupts?

ORCC #$50 (binary 01010000).

* What command specifically enables (turns on) both interrupts?

ANDCC #$AF (binary 10101111).

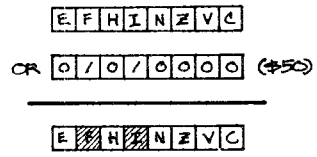
* Three things happen when an interrupt occurs. What are they?

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in reponse to the interrupt.

* For purposes of clarity and simplicity, this session assumes that Morse Code is a maximum of 5 beeps long. For letters and numbers this is true, but punctuation requires 6 beeps. These exceptions will be handled by modifications to the program in the next session. Five beeps are assumed to demonstrate the dramatic simplicity of code translation. Since code translation (ASCII to IBM's EBCDIC, Baudot to ASCII, printer translations, etc.) is an important part of machine-language programming, learning to do it the simplest way is important.

carry, overflow and "entire flag" would be preserved, but bits 4 and 6, the IRQ and fast IRQ interrupts, would be changed. To set interrupts on, then, AND with hex **$AF**; to turn interrupts off, OR with hex **$50**. Much more later.

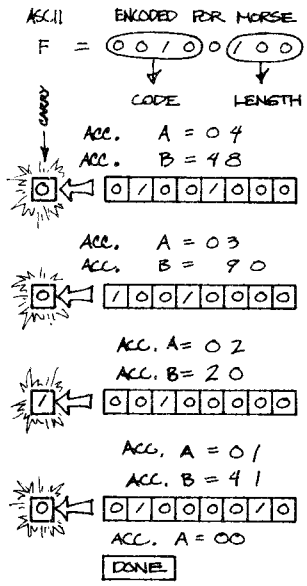
Discussing interrupts has taken this lesson well out of its way. The topic was timing, and specifically, the timing necessary to produce pulses of sound in a known order, with a known pitch, and at a known speed. I'll turn back to that now.



Morse Code was a brilliant invention. It provided a compact method of transmitting letters. It was fast, because the most-used letters contained less beeps than the least-used letters. It accommodated all physical talents because trained operators could send and receive at high speed, whereas novices could still be understood at only a few words per minute. The compactness of Morse Code, however, increases the programming difficulty for us. The letter E, a single dit, contrasts with the letter Z, dah dah dit dit.

For this program, the cross-listing of ASCII codes and Morse Code has to provide two kinds of information: the pattern of dits and dahs, and also the total number of beeps in the letter. The longest character has five beeps, which could be stored as five bits in a byte . . . dits could be represented by zeros and dahs by ones. The remaining three bits could be used to indicate the length of the Morse character. One byte might do the job.

The next question is how to arrange those bits within the byte. The dit-dah pattern could go on either side of the byte, as could the number of beeps. But one arrangement makes special sense. Recall that in an earlier lesson, a binary-to-ASCII conversion was performed. It was always necessary to make sure the nybble was to the right side of the byte to be in the proper form. That's the case here, too. By keeping the rightmost three bits reserved for the length of the Morse Code, the only work you need do is mask the leftmost five bits to retrieve the original number.



Follow me in the book for this. The letter S is dit-dit-dit. According to my suggestion, dit-dit-dit becomes binary **000**. The length is three beeps, so the length is binary **011**. Place the beeps at the left and the length at the right and you've got the composite byte **000 00 011**. By contrast, the letter O is dah-dah-dah. It translates into **111** for the code, and again to **011** for the length. The composite code is **111 00 011**.

But even better is what you can do with the encoded beep information at the left of the byte. By rotating the byte to the left, the beep bits drop into the carry flag in head-first order. Dit-dah-dah-dit, represented by **0110**, rotates left and falls into the carry flag in the precise order **0-1-1-0**, or dit-dah-dah-dit. By using the carry flag as a condition for program branching, the process is assured. The program can branch-on-no-carry to a "dit"-length beep, and branch-

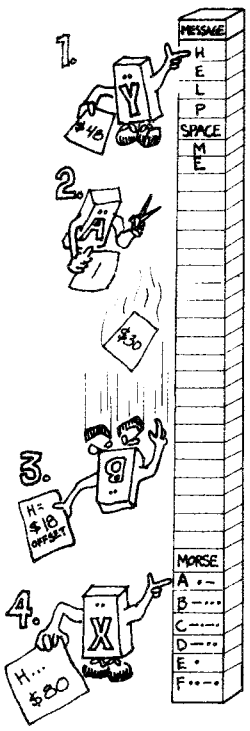
on-carry to a "dah" length beep.

In actual program form, the message SOS might be encoded like this:

ASCII Letter	Morse Code	ASCII Code	Morse Binary	Morse Hex
S	...	$53	%00000011	$03
O	---	$4F	%11100011	$E3
S	...	$53	%00000011	$03

I'd like you to take a break now and draw up a chart of all the ASCII characters and their Morse Code equivalents, as shown in the sample above. This will give you an intimate sense for the way in which this code is being assembled.

> On a sheet of paper, list the characters in ASCII order, their Morse Code equivalents, their ASCII codes, their Morse binary encoding, and their encoded Morse hexadecimal representation. When you have completed the sheet, return to the tape.



Now you've got a complete cross-reference table in hand, and you understand the general workings of the program you've got to create. Let me review the structure so far.

1. Pluck an ASCII value from the message.

2. Find the encoded Morse equivalent in the table.

3. Use the length information in the rightmost three bits as a counter.

4. Shift the leftmost five bits into the carry flag.

5. Transmit dits or dahs based on the carry flag, and for the number of beeps held by the counter.

6. Pick up the next letter and continue.

This looks like a reasonable structure; it should resolve into this simplified program (follow me in the book):

```
START  LDX   MORSE    * Encoded Morse in memory
       LDY   TEXT     * ASCII message in memory
AGAIN  LDB   ,Y+      * Get ASCII, point to next
       SUBB  OFFSET   * Strip ASCII offset
       LDA   B,X      * B+X = Morse table position
       TFR   A,B      * Save encoded Morse
       ANDA  $07      * Keep the code length
```

* What do ASCII codes $20 through $3F represent?

Numbers, symbols and punctuation.

* What do ASCII codes $40 through $5F represent?

Uppercase letters.

* The following questions deal with the specific program being created in this session.

* In the structure chosen for this example, where is the Morse Code length information stored?

In the rightmost three bits of a byte.

* Where is the actual Morse Code pattern stored?

In the leftmost five bits of a byte.

* How is the length information retrieved?

By masking the byte with binary 00000111.

* If the byte is in the A accumulator, what instruction is used to mask in the right three bits?

ANDA #$07

* "Dit" and "dah" are represented by what information?

"Dit" is zero, "dah" is one.

* Where is the "dit" and "dah" information stored?

In the leftmost five bits of a byte.

* How is the information retrieved?

By rotating the bits leftward into the carry flag.

*** What is the advantage to having the code length on the right side of the byte?**

It only needs to be masked, not shifted, to become the correct value.

*** What is the advantage to having the code beeps on the left side of the byte?**

There are two advantages: they are in place to be rotated left into the carry flag, and they are in the correct order from first to last as they are rotated into the carry flag.

*** The letter S is dit-dit-dit. What is its length in binary?**

Its length is three beeps, 011 binary.

*** The letter S is dit-dit-dit. What is the binary equivalent of its beeps?**

The binary equivalent of the beeps is 000.

*** What is the complete encoded byte for letter S, pattern dit-dit-dit?**

Pattern 000 plus two unused bits (00) plus the length 011. The result is 00000011.

*** What is the hex equivalent of binary 00000011?**

Binary 00000011 is hex $03.

*** Name the timing considerations needed for Morse Code.**

The length of the "dit", the length of the "dah", the length of silences, and the frequency of the beep.

```
NEWBIT ROLB          * Drop into carry
       BCS   JUMP1   * On C=1, do dah, do dah
       JSR   DIT     * On C=Ø, go do the dit
       BRA   NEXT    * .. and go past
JUMP1  JSR   DAH     * Here's the dah to play
NEXT   DECA          * Length = Length-1
       BNE   NEWBIT  * Next bit if Length <> Ø
       JMP   AGAIN   * Back for next character
```

There are a few things missing from this structure. As shown, there are no spaces obvious between letters or words. Even if silences were included in the beep routines, there wouldn't be any break between streams of beeps. So silence must be added. And then there's the question of what to do when the message is finished. In my example, the transmission continues right through memory. It's got to be made to stop. To achieve a pause, I've selected a yet-to-be-written subroutine called QUIET. As for the message end, the greater-than and less-than characters aren't present in the Morse Code system. I've decided to use the greater than sign to indicate "end of message," and the less-than sign to mean "repeat message." With that in mind, I've got a program for you to load.

Program #22, 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.

List this program screen by screen using the P command. You'll find that I put the message at **$3000**, the Morse Code table at **$2F00**, and the program itself at **$2E00**. Examine this program carefully, and see if its compactness makes sense. Also, check your handwritten Morse table against mine. All that's left to write are the dit, dah, and silence subroutines. Till next time.