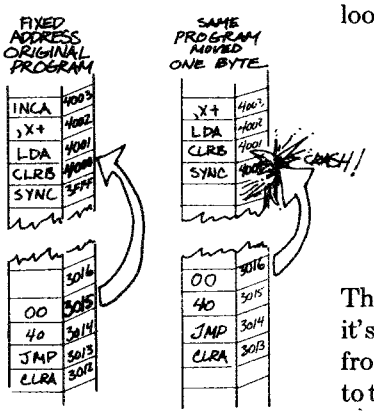
18.

Have you ever typed in a long assembly language program listing from a magazine, accepting on faith that it would work on your Color Computer? And then finding out that your XYZ disk system or your Apex memory dewormer was already using that area of memory? Within certain limitations, that inflexible approach to memory use isn't necessary any more. Utility programs — especially those in semi-permanent installations such as the XYZ disk or Apex dewormer — should be able to be moved to other areas of memory and still perform their advertised functions. Until the introduction of the 6809, microprocessors couldn't offer this as a standard feature... a feature known as Position Independent Programming. Your Color Computer can do it. Position Independent Programming is the topic of this session.

To understand position independence, you have to understand the limitations of position dependence. Have a look at the program in the book; the mnemonics read:




```
1000  8E   1234          LDX  #$1234
1003  108E 5678          LDY  #$5678
1007  B6   FF20  LOOP    LDA  $FF20
100A  27   03            BEQ  LATER
100C  7E   1007          JMP  LOOP
100F  7F   0001  LATER   CLR  $0001
```

There's nothing especially useful about this program, but it's good enough code. The A accumulator is being loaded from what looks like an input port address, and branching to the label LATER if the loaded value is zero. If it's not, the program jumps back to the position marked LOOP.

But what if you needed to move this program from address **$1000** to, for example, address **$2000**? Well, if you were the programmer, you would simply load the source code into EDTASM+ and re-assemble it at the new origin. But if you had purchased the program and you didn't know its structure or contents, but nevertheless needed to move the binary code from **$1000** to **$2000**, something unhappy I sigh at the prospect of having to disassemble, examine and relocate some assembly language applications programs -- spreadsheets are one example -- faced with their enormous size and complexity. This usually happens when I want to tiptoe around some special printer or video driver I've created. With 6809 programs I've had the chance to be pleasantly surprised, since some not only can be located easily in other areas of memory, they automatically relocate themselves to respect memory limits and other configurations you've set ahead of time. Machine language programs which run independent of their position in memory is the exciting goal of this session.

* What is an addressing mode?

The way a machine language program gets its information.

* What addressing mode is JMP $1234?

Extended addressing.

* What addressing mode is BRA LOOP?

Relative addressing.

Program counter relative

* Relative addressing is relative to what?

The program counter (PC).

* How does BRA $FE differ from JMP $3456 if both instructions begin at address $3456?

They differ in that BRA is 2 bytes and relative addressing, whereas JMP is 3 bytes and extended addressing.

* How is BRA $FE similar to JMP $3456 if both instructions begin at address $3456?

Both are endless loops.

* Is BRA $FE an endless loop if it appears at address $3455?

Yes.

* Is JMP $3456 an endless loop if it appears at address $3455?

No.

* What happens to JMP $3456 if it is moved to address $3455?

The desired opcode JMP ($7E) is now at $3455. The program counter points to address $3456 where it finds $34 56 instead of $7E. $34 56 isn't an instruction, but the processor thinks it is, executing $34 56 -- PSHS U,X,A,B. Crash!

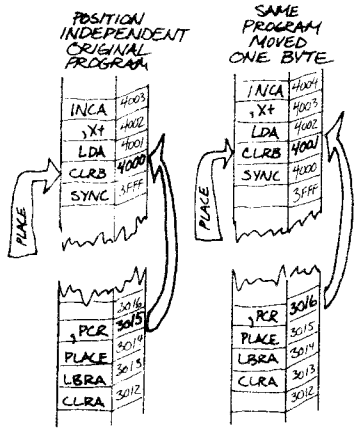
* What do mnemonics BEQ and BNE mean?

Branch if equal to and branch if not equal to.

* What do mnemonics BCC and BCS mean?

Branch on carry clear and branch on carry set.

would occur. Everything in the program would seem perfect until it reached that jump to label LOOP. As far as the binary code is concerned, that jump is specifically to address **$1007**. **$1007** is an absolute, fixed address; with the program now residing at **$2000**, trouble would be on the way. By contrast, the program branch to label LATER is relative addressing . . . the branch is measured from the current position of the program counter. Do you see that? JMP goes to a known, numbered, fixed memory location; BEQ moves to a new position relative to wherever the program is now.



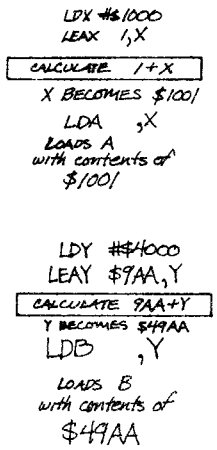
Now, I did use JMP in this example when I could easily have used branch always, BRA. But what if the jump were to an address 5,000 addresses away? An ordinary branch can't move that far, since it's limited to relative movement between +127 and –128. And what about subroutines? The opcode JSR also requires a fixed address. And then there's always the problem of loading X and Y registers with the locations of important tables of information found within the limits of the program. How can these memory locations be identified if not by their fixed locations? Those are the frustrating questions of position independence: how to avoid specifying a fixed, numerical address anywhere in the program.

Well, you can probably guess that I wouldn't be asking those rhetorical questions if I didn't already have an answer. And you're right. The 6809 commands JMP and JSR can be cashed in for the 6809's flexible Branch and Long Branch commands. Not only can you execute long branches to any relative position throughout all of memory, but you can perform long branches to subroutines in any relative position throughout memory. And those load immediate instructions can be cashed in for what's known as "program counter relative" indexing.

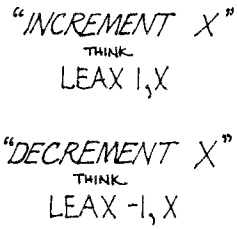
The price you pay for these relative branches or indexings is an additional clock cycle or two, plus a slightly different process of thinking. Everything can become relative to the program counter, not just short and long branches, but even loads and stores. Loads and stores can make use of the special ",PCR" version of the indexed addressing mode.

Before I get carried away with the excitement of generalities, I want you to do a little reading. Open your MC6809E data book, turn to page 17, and read the section headed "Program Counter Relative." Also read page 18, the heading "LEAX/LEAY/LEAU/LEAS." Finally, turn to page 32 and read the summary of the 6809's short and long branch instructions.

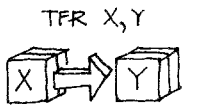
Turn to the MC6809E data book, page 17, and read the section headed "Program Counter Relative." Also turn to page 18, and read the section headed "LEAX/LEAY/LEAU/LEAS." Finally, turn to page 32 and read the summary of the 6809's short and long branch instructions. Return to the tape when you have completed the reading.



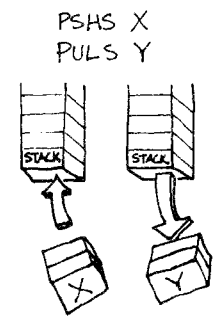
Let me start with the LEA instructions, which are easier to use than to describe; you can be looking at page 18 as I talk. LEA (Load Effective Address) is really no mystery, it's just a highly jargonized name for an old, familiar concept. Here's how LEA came clear to me: There exist no unique increment or decrement instructions for the 16-bit X or Y registers in the 6809. Considering how often I wanted to move these registers forward or back in memory, I thought this might be a serious deficiency in the 6809's capability. Sure, you know that there are automatic increment and decrement modes, but these require loading or storing information to get them to work. So I spent some time cracking my brains over LEAX and LEAY.



I discovered that Increment X is actually **LEAX 1,X**... that is, make X become X with an offset of 1. Decrement X, then, must be **LEAX -1,X**. It seemed clumsy then, but not now. Maybe these are a little less easy to think of or use than a straightforward increment or decrement, but they are many times more flexible. If **LEAX 1,X** makes X become X+1, then **LEAX 2,X** makes X become X+2. You're no longer limited to simple increments or decrements. **LEAX -40,X** makes X equal X–40. **LEAY 12345,Y** makes Y equal Y+12345. That was the key. I began to understand that the clumsy phrase "load effective address" was a jargon-filled way of saying the same thing that "LET" says in BASIC. Whereas BASIC would say LET Y = Y+150, the 6809 assembly language says **LEAY 150,Y** . . . load Y with the effective address 150+Y.



But there's more. Not only can X=X+10 by writing **LEAX 10,X**, but X can equal Y+10 by writing **LEAX 10,Y**. . . or Y can equal S–50 by writing **LEAY -50,S** . . . or U can equal X by writing **LEAU 0,X**. In fact, depending on your requirements, the 6809 processor offers three different ways of making one 16-bit register equal another: you've got **TFR X,Y**. Then there's **PSHS X** followed by **PULS Y**. And then you can **LEAX 0,Y**.



Here's more about Load Effective Address. You can use the A, B or combination D accumulators as variable offsets. For example, X can be made equal to A plus X by writing **LEAX A,X**.

But by far the most versatile and powerful application of the LEA instructions is in the writing of position independent programs. In the programs I've presented so far, I've always loaded the X or Y registers with specific values. For example, in the Life program that was

* What is the branching range of BRA (and other branch instructions)?

PC-128 to PC+127 (PC-$80 to PC+$7F).

* What does LBRA mean?

Long branch always.

* What is the branching range of LBRA (and other long branch instructions)?

* PC-32768 to PC+32767 (PC-$8000 to PC+$7FFF).

* What addressing mode is BEQ LOOP?

Relative addressing.

* What addressing mode is LBEQ LOOP?

Relative addressing.

* What does LEA mean?

LEA means Load Effective Address.

* What is the effect of LEAX 1,X?

X becomes X+1.

* What is the effect of LEAX $45,X?

X becomes X+$45.

* What is the effect of LEAX 1,Y?

X becomes Y+1.

* What is the effect of LEAX -5,Y?

X becomes Y-5.

* What is the effect of LEAY 12345,Y?

Y becomes Y+12345 (Y+$3039).

Simple branches

* If A is $32 and X is $1000, what is the effect of LEAX A,X?

X becomes X+A, that is, X becomes $1032.

* If X = $1000, give the value of Y after:
TFR X,Y

Y becomes $1000.

* If X = $1000, give the value of Y after:
PSHS X
PULS Y

Y becomes $1000.

* If X = $1000, give the value of Y after:
LEAY 0,X

Y becomes $1000.

* If X = $1010, give the value of Y after:
LEAY -16,Y

Y becomes $1000.

* What does LEA mean?

LEA means Load Effective Address.

* What does ",PCR" mean?

",PCR" means program counter relative mode.

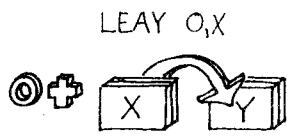
* If the instruction LDX #ARITH1 is found at address $1000, and label ARITH1 points to $2000, what is X after the instruction is executed?

X points to $2000.

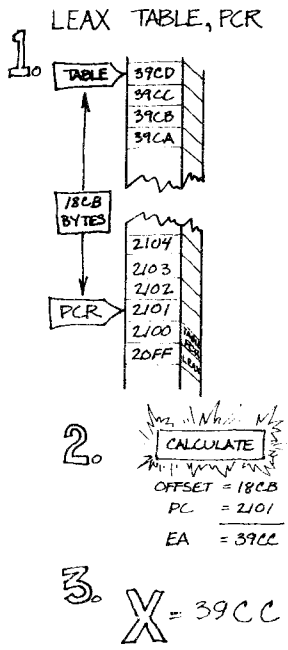
* If the instruction LDX ARITH1,PCR is found at address $1000, and label ARITH1 points to $2000, what is X after the instruction is executed?

X points to $2000.

completed in the last session, you remember that the X register was pointed to a table of information by loading the X register with the actual address of the table. I wrote **LDX #TABLE**. But there's another way, a position independent way.



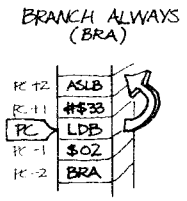
I might instead have written **LEAX TABLE,PCR**. That's **LEAX TABLE,PCR**. And that says "Load X with the effective address calculated from the distance between the present position of the program counter and the address of the table." In other words, I know the distance from here to where I'm going. By giving that distance to the 6809, it can calculate the resulting address, and give that result to the X register.



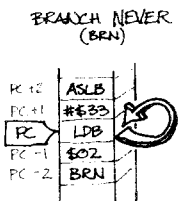
No longer are you constrained to a fixed address. Instead of demanding to know, "where is it?", the 6809 need only ask "how far is it from here?". I'll get back to Load Effective Address; in the meantime, just remember that when you see LEAX, think LET X. You see **LEAY 10,Y** and you think LET Y be 10 plus Y. Purists might want my head for that, but I'll risk it. When you see LEA, think LET.

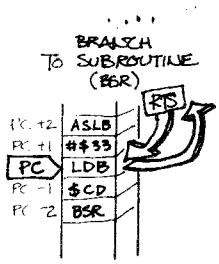
Among the other position-independent commands are the branches. You've been using the branches since early on in this course, but I've never given them any formal time. I'll make up for that now.

Like the program counter relative instructions, the branches are also based on "how far from here?" rather than "where?". In all, there are 62 variations of relative branches, depending on how you think of them. Turn to page 32 of the MC6809E data book. You'll see the branch instructions in four groups: simple, simple conditional, signed conditional, and unsigned conditional. Some overlap, serving dual purposes. I'm going to describe the short branches, but keep in mind that the long branches are identical in principle and application. The only difference is that the short branches reach a span of 256 bytes, and the long branches reach a span of 65,536 bytes.

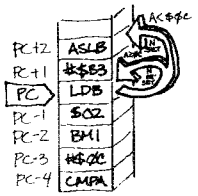


Simple branches are just that. When the instruction decoder finds a simple, it follows the command, calculates the new address, and hands it to the program counter. These three are branch always (BRA), branch never (BRN) and branch to subroutine (BSR). Two of these make sense; but what about "branch never"? "Branch never" is one of those delightful bizarrities of computer logic. "Branch never" exists as a default of the processor's architecture. All branches have what are called true and false versions; branch always is the true version, so "branch never" is the false version. Branch always makes the branch, very much like the command JMP. "Branch never" continues with the main program flow. But keep it in mind; it's surprisingly useful. Should you be doing critical timing where every machine byte and clock cycle counts, remember that no operation (NOP) uses one byte and 2 cycles; "branch never" has the effect of a NOP, but it uses two bytes and 3 cycles; and long "branch never" also has the effect of a NOP, but it uses 4 bytes and 5 cycles.

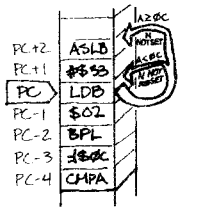




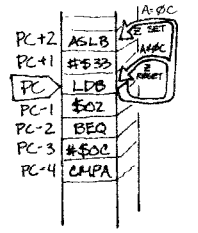
BRANCH ON MINUS (BMI)



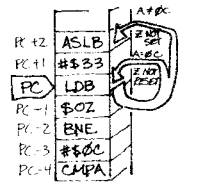
BRANCH ON PLUS (BPL)

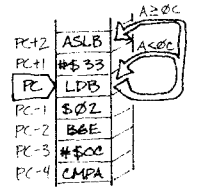


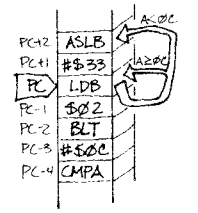
BRANCH ON EQUAL (BEQ)



BRANCH ON NOT EQUAL (BNE)







Enough of the simple branches; on to the simple conditional branches. These are changes of program flow conceived of as direct responses to the condition codes.

1. Branch on minus and branch on plus are in response to the state of the negative (N) flag.

2. Branch on equal and branch on not equal are in response to the state of the zero (Z) flag.

3. Branch on overflow set and branch on overflow clear respond to the state of the overflow (V) flag.

4. Finally, branch on carry set and branch on carry clear respond to the state of the carry (C) flag.

Those eight conditional branches should make sense to you, since you've used most of them in programming already.

The signed and unsigned conditional branches take account of not only the flags but also the type of arithmetic being used, in order to produce a composite result and make a branching decision. The signed conditional branches assume that you are using signed arithmetic, that is, where you are thinking in terms of positive and negative, so that the most significant bit is important to the calculation. There are three types of signed conditional branch, arranged five ways:

1. Branch on greater than (BGT), and its opposite, branch on less than or equal to (BLE). Remember that in signed arithmetic, **$01** is greater than **$FE**, that is, 1 is greater than –1.

2. The complementary instructions to the previous ones are branch on greater than or equal to (BGE) and branch on less than (BLT).

3. Signed branches also make use of the familiar branch on equal (BEQ) and branch on not equal (BNE).

4 and 5. The final two pairs of branches are identical to the first to pairs, but are conceived in reverse. At the end of this lesson, take the time to examine the four tables at the bottom of page 32 of the data booklet, and try to clarify how the pair "branch on greater than"/"branch on less than or equal to" is different in conception from "branch on less than or equal to"/"branch on greater than".

The remaining branch types are the unsigned conditional branches. These are effectively identical to the previous

* What does BSR mean?

BSR means Branch to subroutine.

* What do mnemonics BGT, BGE, BLT and BLE mean?

Branch on greater than, branch on greater than or equal to, branch on less than, and branch on less than or equal to.

* What do mnemonics BRA and BRN mean?

Branch always and branch never.

* In unsigned arithmetic, which is the higher number, $7F or $00?

$7F is a higher number than $00.

* In unsigned arithmetic, which is the higher number, $AA or $55?

$AA is a higher number than $55.

* In signed arithmetic, which is the greater number, $AA or $55?

$55 (being positive) is greater than $AA (being negative).

* In signed arithmetic, which is the greater number, $FF or $00?

$00 is greater than $FF (-1).

* What specific kind of instruction is BGT (branch on greater than)?

BGT is a signed conditional branch.

* What specific kind of instruction is BHS (branch on higher than or same as)?

BHS is an unsigned conditional branch.

Selecting branches

* If A contains $FF and is compared to memory containing $00, would the branch BGT be taken or not? Why?

It would not be taken because $FF (decimal -1) is less than $00, and BGT is a signed conditional branch.

* If A contains $FF and is compared to memory containing $00, would the branch BHS be taken or not? Why?

The branch would be taken because $FF (decimal 255) is higher than $00, and BHS is an unsigned conditional branch.

* What addressing mode are BHS and BGT?

Relative addressing.

* What addressing mode is JMP $1234?

Extended addressing.

* What addressing mode is JMP ($1234)?

Extended indirect addressing.

* What does the mnemonic LBLO mean?

Long branch if lower than.

* What addressing mode is this?

Relative addressing.

* What is the branching range of BLO?

The range is -128 ($80) to +127 ($7F) relative to the program counter.

* What is the branching range of LBLO?

The range is -32768 ($8000) to +32767 ($7FFF), relative to the program counter.

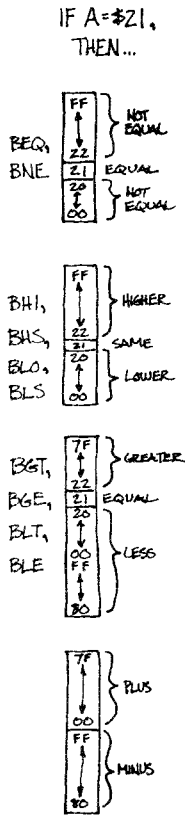
ones, but negativeness or positiveness do not affect the result. These branches are:

1. Branch on higher than (BHI), and its opposite, branch on lower than or same as (BLS). In unsigned arithmetic, **$FE** is greater than **$01**, that is, 254 is greater than 1.

2. Branch on higher than or same as (BHS), and its opposite, branch on lower than (BLO).

3. The familiar branch on equal (BEQ) and branch on not equal (BNE) are also part of the unsigned set of branches.

4 and 5. Finally, there are the inverse pairs of the first sets of conditions. Again, examine these tables at the end of the lesson.



So how do these all fit together? How do you choose among simple conditional, signed conditional, and unsigned conditional branches? Here's how:

- If you're using the flags directly, such as with rotations, yes/no comparisons, etc., use the simple conditional branches. If you're thinking about the condition codes per se, then you want to use simple conditional.

- If you're doing arithmetic, such as creating mathematical subroutines, or if you're using numbers transferred from BASIC, use signed conditional branches. Real numbers are positive and negative, so use signed conditional branches when doing that kind of math.

- If you're making a series of value comparisons or checking table entries, then use the unsigned conditional branches. These are similar to the simple conditional branches, except they allow you a little more flexibility or programming compactness.

Some experimenting will make the choices clear. I've got a program I think you'll like. Get your computer on and up in Extended BASIC. When you're ready, type and enter this BASIC line; follow in your book:

```
PCLEAR8:PMODE4,1:PCLS:PMODE4,5:PCLS:CLOADM:EXEC
```

Your computer will be ready and searching for an object code program. It's coming up.

Program #28, an object code program. Turn on the power to 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, type EXEC and press ENTER. The program will execute automatically. If an I/O error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

BASIC started by clearing an area of graphics memory, so what you should be seeing is a clean high-resolution graphics screen with a narrow, random-looking band of colors walking down the screen from top to bottom. At the same time, a continuous tone is coming from the loudspeaker. The tone hiccups each time the colored band moves down the screen.

Before you sigh "so what" to yourself, let me tell you what you're looking at. The band of random color isn't random at all. It's the program. The program itself is being displayed as if it were screen information. That shouldn't be a surprise, since memory is memory so far as the microprocessor is concerned. But it can be disconcerting to actually snoop into the program's private memory lair.

Now to my point. This band of color is MOVING. The program is producing a tone, then moving itself, erasing its trail, and re-executing in a new position in memory. Eventually, the loudspeaker will let out a strangled squawk and probably return an "OK" to your screen, as the moving program crashes into the un-writable BASIC ROM.

This is a completely position-independent program. When you're ready, you can load the assembly source code and have a look. I'll be back for the next lesson and a complete walk-through of this program, and a re-explanation and summary of the process of position-independent code. Enjoy this one.

Program #29, 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.

```
1000                    00100          ORG     $1000
                        00110 *
            FF20        00120 SPORT    EQU     $FF20
            00AA        00130 DIFFER   EQU     LAST-FIRST
                        00140 *
                        00150 * DISABLE THE INTERRUPTS
1000 1A     50          00160 FIRST    ORCC    #$50
                        00170 *
                        00180 * OPEN THE SOUND LATCH
1002 86     3C          00190          LDA     #$3C
1004 B7     FF23        00200          STA     $FF23
                        00210 *
```

* How many groups of branches are there?

There are four groups of branches.

* What are the four kinds of branches?

Simple branches, simple conditional branches, unsigned conditional branches, and signed conditional branches.

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

Program #29

```
                        00220 * SELECT VIDEO ADDRESS
1007 C6   07            00230          LDB     #$07
1009 8E   FFC6          00240          LDX     #$FFC6
100C A7   81            00250 VIDEO    STA     ,X++
100E 5A                 00260          DECB
100F 26   FB            00270          BNE     VIDEO
1011 B7   FFCD          00280          STA     $FFCD
                        00290 *
                        00300 * SELECT GRAPHICS MODE
1014 B7   FFC5          00310          STA     $FFC5
1017 B7   FFC3          00320          STA     $FFC3
101A B7   FFC0          00330          STA     $FFC0
                        00340 *
                        00350 * SELECT COLOR SET, MODE
101D 86   C7            00360          LDA     #$C7
101F B7   FF22          00370          STA     $FF22
                        00380 *
                        00390 * ERASE PREVIOUS PROGRAM
1022 C6   AA            00400 ERASE    LDB     #DIFFER
1024 30   8C D9         00410          LEAX    FIRST,PCR
1027 30   89 FF56       00420          LEAX    -DIFFER,X
102B 4F                 00430          CLRA
102C A7   80            00440 KLEEN    STA     ,X+
102E 5A                 00450          DECB
102F 26   FB            00460          BNE     KLEEN
                        00470 *
                        00480 * BEEP FOR ALL TO HEAR
1031 8D   12            00490          BSR     BEEP
                        00500 *
                        00510 * TRANSFER PROGRAM AHEAD
1033 C6   AA            00520          LDB     #DIFFER
1035 30   8C C8         00530          LEAX    FIRST,PCR
1038 31   8D 006E       00540          LEAY    LAST,PCR
103C A6   80            00550 LOOP     LDA     ,X+
103E A7   A0            00560          STA     ,Y+
1040 5A                 00570          DECB
1041 26   F9            00580          BNE     LOOP
                        00590 *
                        00600 * AND GO TO MOVED PROGRAM
1043 20   65            00610          BRA     LAST
                        00620 *
1045 86   FF            00630 BEEP     LDA     #$FF
1047 34   02            00640 REBEEP   PSHS    A
1049 86   3E            00650          LDA     #$3E
104B 30   8D 001C       00660          LEAX    WAVES,PCR
104F E6   86            00670 WAVER    LDB     A,X
1051 58                 00680          ASLB
1052 58                 00690          ASLB
1053 F7   FF20          00700          STB     SPORT
1056 8D   09            00710          BSR     DELAY
1058 4A                 00720          DECA
1059 26   F4            00730          BNE     WAVER
105B 35   02            00740          PULS    A
105D 4A                 00750          DECA
105E 26   E7            00760          BNE     REBEEP
1060 39                 00770          RTS
                        00780 *
1061 34   02            00790 DELAY    PSHS    A
1063 86   06            00800          LDA     #$06
1065 4A                 00810 DLOOP    DECA
1066 26   FD            00820          BNE     DLOOP
1068 35   02            00830          PULS    A
106A 39                 00840          RTS
                        00850 *
106B      1F1C          00860 WAVES    FDB     $1F1C
106D      1916          00870          FDB     $1916
106F      1310          00880          FDB     $1310
1071      0D0B          00890          FDB     $0D0B
1073      0806          00900          FDB     $0806
1075      0403          00910          FDB     $0403
1077      0201          00920          FDB     $0201
1079      0000          00930          FDB     $0000
107B      0000          00940          FDB     $0000
107D      0001          00950          FDB     $0001
107F      0204          00960          FDB     $0204
1081      0608          00970          FDB     $0608
1083      0A0C          00980          FDB     $0A0C
1085      0F12          00990          FDB     $0F12
1087      1417          01000          FDB     $1417
1089      1B1E          01010          FDB     $1B1E
108B      2124          01020          FDB     $2124
108D      272A          01030          FDB     $272A
108F      2D30          01040          FDB     $2D30
```

```
1091       3235      01050            FDB      $3235
1093       3739      01060            FDB      $3739
1095       3A3C      01070            FDB      $3A3C
1097       3D3E      01080            FDB      $3D3E
1099       3E3E      01090            FDB      $3E3E
109B       3E3E      01100            FDB      $3E3E
109D       3D3C      01110            FDB      $3D3C
109F       3B39      01120            FDB      $3B39
10A1       3735      01130            FDB      $3735
10A3       3330      01140            FDB      $3330
10A5       2E2B      01150            FDB      $2E2B
10A7       2825      01160            FDB      $2825
10A9       22        01170            FCB      $22
                     01180 *
           10AA      01190 LAST       EQU      *
                     01200 *
           1000      01210            END      FIRST
00000 TOTAL ERRORS
```

19.

Welcome back. During this session I want to review the concept of position independent programming, and to take you through the self-moving, position-independent program from the end of the last lesson. Get that source code loaded again.

Program #29, 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.

```
1000                    00100          ORG      $1000
                        00110 *
             FF20       00120 SPORT    EQU      $FF20
             00AA       00130 DIFFER   EQU      LAST-FIRST
                        00140 *
                        00150 * DISABLE THE INTERRUPTS
1000 1A      50         00160 FIRST    ORCC     #$50
                        00170 *
                        00180 * OPEN THE SOUND LATCH
1002 86      3C         00190          LDA      #$3C
1004 B7      FF23       00200          STA      $FF23
                        00210 *
                        00220 * SELECT VIDEO ADDRESS
1007 C6      07         00230          LDB      #$07
1009 8E      FFC6       00240          LDX      #$FFC6
100C A7      81         00250 VIDEO    STA      ,X++
100E 5A                 00260          DECB
100F 26      FB         00270          BNE      VIDEO
1011 B7      FFCD       00280          STA      $FFCD
                        00290 *
                        00300 * SELECT GRAPHICS MODE
1014 B7      FFC5       00310          STA      $FFC5
1017 B7      FFC3       00320          STA      $FFC3
101A B7      FFC0       00330          STA      $FFC0
                        00340 *
                        00350 * SELECT COLOR SET, MODE
101D 86      C7         00360          LDA      #$C7
101F B7      FF22       00370          STA      $FF22
                        00380 *
                        00390 * ERASE PREVIOUS PROGRAM
1022 C6      AA         00400 ERASE    LDB      #DIFFER
1024 30      8C D9      00410          LEAX     FIRST,PCR
1027 30      89 FF56    00420          LEAX     -DIFFER,X
102B 4F                 00430          CLRA
```

The position-independent program really isn't all just tricks and gimmicks. Its real purpose is to make the machine code "transportable". BASIC is transportable; you don't need to load it to a specific memory location. You just load and run. High-level languages have to work that way, but machine language had a hard time ... until the 6809.

* What is a position independent program?

A program designed to run correctly no matter where it is located in memory.

Program #29 reprise

```
102C A7   80        00440 KLEEN   STA     ,X+
102E 5A             00450         DECB
102F 26   FB        00460         BNE     KLEEN
                    00470 *
                    00480 * BEEP FOR ALL TO HEAR
1031 8D   12        00490         BSR     BEEP
                    00500 *
                    00510 * TRANSFER PROGRAM AHEAD
1033 C6   AA        00520         LDB     #DIFFER
1035 30   8C C8     00530         LEAX    FIRST,PCR
1038 31   8D 006E   00540         LEAY    LAST,PCR
103C A6   80        00550 LOOP    LDA     ,X+
103E A7   A0        00560         STA     ,Y+
1040 5A             00570         DECB
1041 26   F9        00580         BNE     LOOP
                    00590 *
                    00600 * AND GO TO MOVED PROGRAM
1043 20   65        00610         BRA     LAST
                    00620 *
1045 86   FF        00630 BEEP    LDA     #$FF
1047 34   02        00640 REBEEP  PSHS    A
1049 86   3E        00650         LDA     #$3E
104B 30   8D 001C   00660         LEAX    WAVES,PCR
104F E6   86        00670 WAVER   LDB     A,X
1051 58             00680         ASLB
1052 58             00690         ASLB
1053 F7   FF20      00700         STB     SPORT
1056 8D   09        00710         BSR     DELAY
1058 4A             00720         DECA
1059 26   F4        00730         BNE     WAVER
105B 35   02        00740         PULS    A
105D 4A             00750         DECA
105E 26   E7        00760         BNE     REBEEP
1060 39             00770         RTS
                    00780 *
1061 34   02        00790 DELAY   PSHS    A
1063 86   06        00800         LDA     #$06
1065 4A             00810 DLOOP   DECA
1066 26   FD        00820         BNE     DLOOP
1068 35   02        00830         PULS    A
106A 39             00840         RTS
                    00850 *
106B      1F1C      00860 WAVES   FDB     $1F1C
106D      1916      00870         FDB     $1916
106F      1310      00880         FDB     $1310
1071      0D0B      00890         FDB     $0D0B
1073      0806      00900         FDB     $0806
1075      0403      00910         FDB     $0403
1077      0201      00920         FDB     $0201
1079      0000      00930         FDB     $0000
107B      0000      00940         FDB     $0000
107D      0001      00950         FDB     $0001
107F      0204      00960         FDB     $0204
1081      0608      00970         FDB     $0608
1083      0A0C      00980         FDB     $0A0C
1085      0F12      00990         FDB     $0F12
1087      1417      01000         FDB     $1417
1089      1B1E      01010         FDB     $1B1E
108B      2124      01020         FDB     $2124
108D      272A      01030         FDB     $272A
108F      2D30      01040         FDB     $2D30
1091      3235      01050         FDB     $3235
1093      3739      01060         FDB     $3739
1095      3A3C      01070         FDB     $3A3C
1097      3D3E      01080         FDB     $3D3E
1099      3E3E      01090         FDB     $3E3E
109B      3E3E      01100         FDB     $3E3E
109D      3D3C      01110         FDB     $3D3C
109F      3B39      01120         FDB     $3B39
10A1      3735      01130         FDB     $3735
10A3      3330      01140         FDB     $3330
10A5      2E2B      01150         FDB     $2E2B
10A7      2825      01160         FDB     $2825
10A9      22        01170         FCB     $22
                    01180 *
          10AA      01190 LAST    EQU     *
                    01200 *
          1000      01210         END     FIRST
00000 TOTAL ERRORS
```

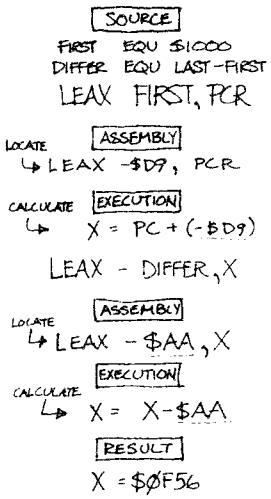
```
BEEP    1045
DELAY   1061
DIFFER  00AA
DLOOP   1065
ERASE   1022
FIRST   1000
KLEEN   102C
LAST    10AA
LOOP    103C
REBEEP  1047
SPORT   FF20
VIDEO   100C
WAVER   104F
WAVES   106B
```

The opening lines of the source code should look familiar to you. Interrupts are disabled to keep the tone pure; the sound latch is opened (recall that process from the Morse Code routine); the video address **$1000** is selected via the SAM registers; high-resolution color graphics, color set, and detail level are selected through an address port. Up to that point, everything is as it has been.

The real differences begin with the routine labeled ERASE. The value identified as DIFFER has been calculated by the assembler from my labels LAST minus FIRST. The first byte of the program I labeled FIRST, and one byte after the last byte I labeled LAST. At the start of the assembly listing, I have the assembler calculate LAST minus FIRST... which is, of course, the length of the entire program. So accumulator B is loaded with the length of the program.

There follow two significant instructions . . .

```
LEAX    FIRST,PCR
LEAX    -DIFFER,X
```



LEAX FIRST,PCR requests that the assembler compute the distance from the program counter to the label FIRST, and make the resultant address available for use by the X register. In other words, after **LEAX FIRST,PCR**, the X register points to the beginning of the program. Then comes the instruction **LEAX -DIFFER,X**. That command instructs the processor to let X equal the present X value minus the value DIFFER. So the effect of those two instructions is to point the X register to a place in memory one program length before the program. Let me go through that one more time. **LEAX FIRST,PCR** is a program-counter relative instruction that calculates the distance between the current position of the program counter and the label FIRST, and assigns the resultant address to register X. Using this technique, X ends up pointing to the start of the program, without ever knowing what absolute address that start actually is until now. After that, **LEAX -DIFFER,X** provides the X register with the effective address X offset by -DIFFER. Let X equal X minus DIFFER. X now points to a location in memory DIFFER places back from its previous position, still without ever knowing the absolute address beforehand. Again: **LEAX FIRST,PCR**. Let X point to the address FIRST places from the program counter. **LEAX -DIFFER,X**. Let X point to the address -DIFFER places away from its previous position. No specific addresses involved . . . position independent . . . program-counter relative.

```
* How many groups of branches
are there?

There are four groups of
branches.

* What are the four kinds of
branches?

Simple branches, simple
conditional branches, unsigned
conditional branches, and signed
conditional branches.

* What is the branching range of
the branch instructions?

The range is -128 ($80) to +127
($7F) relative to the program
counter.

* What is the branching range of
the long branch instructions?

The range is -32768 ($8000) to
+32767 ($7FFF), relative to the
program counter.

* What addressing mode are all
the branches, both long and
short?

Relative addressing.

* Relative addressing is
relative to what?

The program counter.

* What does ",PCR" mean?

Program counter relative.

* What does LEA mean?

LEA means Load Effective
Address.
```

* What is the effect of LEAX 1,X?

X becomes X+1.

* What is the effect of LEAX $45,X?

X becomes X+$45.

* What is the effect of LEAX 1,Y?

X becomes Y+1.

* What is the effect of LEAX -5,Y?

X becomes Y-5.

* If A is $32 and X is $1000, what is the effect of LEAX A,X?

X becomes X+A, that is, X becomes $1032.

* What is the effect of LEAX 1,X?

X becomes X+1.

* What is the effect of LEAX -1,X?

X becomes X-1.

* The 6809 processor provides an INCA command. What is the equivalent of INCX, a fictitious command?

LEAX 1,X

* The 6809 provides a DECA command. What is the equivalent of DECX, a fictitious command?

LEAX -1,X

* If the first byte of a program is labeled START, what is the effect of LEAX START,PCR if the program is ORGed at $1000?

X becomes $1000.

The next four instructions fill up the memory area from –DIFFER,X to FIRST with zeroes; the B register contains DIFFER, the total number of bytes in the program. That is, a block of memory as long as the program from –DIFFER,X to FIRST will be cleared to zero.

Following those contortions is a relative branch to the subroutine BEEP. I'll get back to BEEP in a minute.

After the branch to and back from BEEP, the B register is once more loaded with the program's length. Following that . . .

```
LEAX    FIRST,PCR
LEAY    LAST,PCR
```

Again using the program counter relative technique, the X register is pointed to the beginning of the program, and the Y register is pointed to the byte after the last byte in the program. By means of a standard load-and-store loop — which should be tiresomely familiar by now — the information pointed to by X is transferred to memory pointed to by Y, and both memory pointers are incremented by one. The loop continues until B is decremented to zero. In other words, a copy of the program is made immediately following the end of itself.

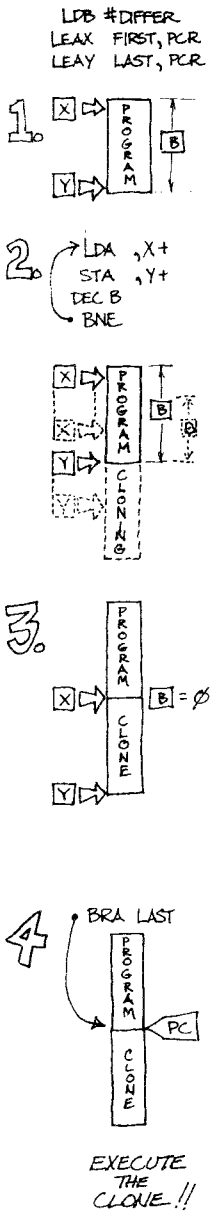
The final instruction is the grabber. The program is told to execute a branch to the label LAST. The LAST has become the FIRST. The program, having just been copied, is born again and seemingly begins anew in a fresh area of memory. It once again sets up the video and sound parameters — a redundant act I included for effect. At this point, the reason for the ERASE routine presented earlier should become clear. ERASE causes the previous program to be cleared out of memory — the program hides its own trail as it beeps and copies itself.

So what you see is a screen full of memory, and revealed on that screen you are watching is a program that beeps, duplicates itself in a new location, branches into its new self, and eradicates its old self.

Chances are you wouldn't ever need to write a program like this. But you might want to write something like the BEEP subroutine, a routine that you can stuff anywhere you like in memory. Have a look at it.

Part of its structure should be familiar. The A register is set up as the length of the beep, and there are values being sent out the sound port to the television speaker. But there's something new. **LEAX WAVES,PCR** (again using program-counter relative addressing) points the X register to a table labeled WAVES. So what's this table?

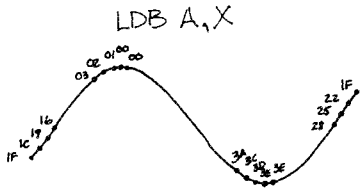
It might look at first like a table of addresses. It isn't. It's a 63-byte reference table . . . these are bytes, not addresses. I just wanted to save myself some typing by compressing them the way you see them. So you can read this table as a



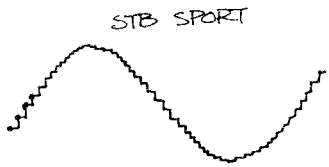
group of 63 bytes: **$1F $1C $19 $16 $13 $10 $0D**, etc. Translated back into the form in which I created them, they read like this:

```
.ØØØ3Ø73
.Ø995276
.1983681
.2952265
.3891352
.4791557
.5643887
.6439825
```

. . . and so forth. It's actually a table of mathematical sines, made positive and multiplied by a constant so that the table falls into the range of positive integers 0 to 63. The reason I've done this is because the Color Computer contains a 6-bit digital-to-analog converter, a circuit which converts a 6-bit binary number into an equivalent voltage. That voltage can be used for a variety of purposes, including the production of sound.



I described this briefly when you were exploring the Morse Code examples. This time you'll be putting it to use. Move back now to the BEEP routine itself. Notice that beginning with the third instruction, the BEEP program loads the A accumulator with **$3E**, points the X register to that table, and then loads the value found at X indexed by A into the B accumulator. The value is shifted to the left (from the low 6 bits to the high 6 bits, where the computer's digital-to-analog converter output happens to be wired). That value is then stored at SPORT, the sound output address in the computer. A brief delay is made, then the next element in the table is acquired and output to the sound port, until all 63 elements have been used up. The routine then loops until 255 repetitions of the table have been output.



The sine wave is the simplest of all musical sounds. By creating a series of numerical values which outline a sine-shaped wave and subsequently putting those values through the computer's 6-bit converter, an equivalent sound wave is produced through the loudspeaker. It sounds like the sine wave it represents.

Take a break now, and make some changes in the subroutine. You can assemble and use the BEEP subroutine separately, if you like. If you use it separately, remember to turn off interrupts by using **ORCC #$50**, and also to turn on the sound latch by storing **$3C** at memory address **$FF23**. I'd like you to play around with the length of the beep (found at line 630 being loaded into the A register), with the frequency of the beep (found in the delay loop at line 800), and with the quality of the sound (by changing the values in the wavetable beginning at line 860). When you're comfortable with how these routines work, thoroughly review both this lesson and the previous one. I'll be back with a summary of position independent programming, and then I'll finish up this session by introducing the remaining 6809E instructions.

```
* If the first byte of a program
is labeled START, what is the
effect of LEAX START,PCR if the
program is ORGed at $1234?

X becomes $1234.

* If the first byte of a program
is labeled START, what is the
effect of LEAX START,PCR if the
program is ORGed at $AAAA?

X becomes $AAAA.

* What addressing mode is LEAX
WAVES,PCR?

Program-counter relative.

* What is a pseudo-op?

An instruction to the
assembler.

* What pseudo-op places a single
byte in memory?

FCB.

* What pseudo-op places two
consecutive bytes in memory?

FDB.

* What pseudo-op places an ASCII
string of characters in memory?

FCC.

* Does the Color Computer have a
digital-to-analog converter?

Yes.

* A digital-to-analog converter
converts what to what?

A binary number to an equivalent
voltage.

* At what memory location is the
Color Computer's digital-
to-analog converter found?

At location $FF22.
```

* How many bits can be sent to the Color Computer's digital-to-analog converter?

6 bits.

* What is the range (in binary, hex and decimal) of the Color Computer's digital-to-analog converter?

Binary 000000 to 111111; hexadecimal $00 to $3F; decimal 0 to 63.

* The Color Computer's digital-to-analog converter ranges from 0 to 5 volts, divided into 64 steps. Zero output is 0/64ths, full output is 64/64ths; that is, it has a step size or resolution of 1/64th of the output. If 000000 is sent to the digital-to-analog converter, what is the output?

000000 is 0/64ths, or 0 volts.

* If 111111 is sent to the digital-to-analog converter, what is the output?

111111 is 63/64ths, or 4.921875 volts.

* If 101010 is sent to the digital-to-analog converter, what is the output?

101010 is 42/64ths, or 3.28125 volts.

* If all the values from 000000 to 111111 and back to 000000 are sent to the digital-to-analog converter, what will a graph of the final voltage output look like?

A triangle.

* If the Color Computer's digital-to-analog converter were 7 bits instead of six, what would be the step size (the resolution)?

1/128th of the output.

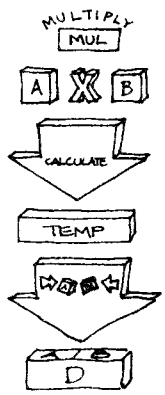
Experiment with the length, pitch and sound quality of the beep in this program. The length of the beep is loaded into the A register in line 630 of Program 29. The frequency of the beep is found in the delay loop in line 800. The wavetable begins at line 860. When you are confident you understand the application of these features, return to the tape.

Position independent programming, then, is the creation of machine language in a way that allows the final assembled binary program to execute anywhere in memory. This quality of position independence is achieved by making all memory pointers, program branches and subroutines relative to the position of the program counter. In that way, the processor never needs to know "where", but only needs to know "how far from here".

Among the commands used with position independent programming are the three dozen variants of the branch (with its 256-byte range) and the long branch (with its 65,536-byte range). Branches come in simple form, where they are always obeyed; in simple conditional form, where their actions depend on the state of specific condition codes; in unsigned conditional form for "higher" and "lower" judgments; and in signed conditonal form for "greater than" and "less than" judgments in with positive and negative arithmetic.

The other commands to achieve position independence are the LEA, or load effective address, group. When used with in program-counter relative form, 16-bit registers can be pointed to any location in memory by virtue of that location's position relative to the current position of the program counter. It's almost mandatory to use an editor/assembler and labels to do this. For the experience, you might try hand-assembling a few LEAX instructions in the program-counter-relative mode.

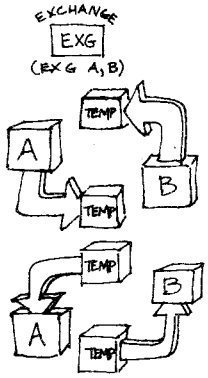
The advantages of position independence are obvious; the disadvantages are a slight increase in the amount of programming code required, and a loss in execution speed. For fast action games and high speed — where position independence is hardly necessary anyway — compact, address-specific programming is adequate and desirable. For utility programs, mathematical subroutines, and other semi-permanent programs (especially those which will be used with other machine-language software), position independence is virtually required.



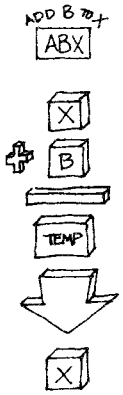
Only a few commands remain in the 6809 instruction set. Some you've come across, and some are brand new to this course. One you've seen is multiply, **MUL**. When **MUL** is executed, the contents of the A accumulator is multiplied by the contents of the B accumulator, and the result is placed in the combined D accumulator. This is an unsigned multiply, meaning the full 8 by 8 bit multiplication is

completed without reference to it being positive or negative. Positive integers are assumed for this multiplication. Although **MUL** takes 11 machine cycles (it is the longest 6809 instruction), it saves the several steps required by other processors, where multiplication is done by many succeeding steps of shifting and adding.

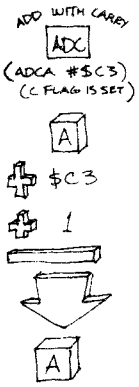
Another you've already seen is no operation, mnemonic **NOP**. The **NOP** has several uses, most frequently as a time-waster for sound, input/output, communication, or other timing loops. The **NOP** takes two cycles to execute, during which no other aspect of the procesors's operation is affected.



Another instruction which you haven't specifically used, but is in a familiar family, is exchange, EXG. Like the transfer (TFR) command, EXG uses an opcode and a postbyte to describe the registers needed. TFR replicates the value in the source register into the destination register. EXG swaps the values in the two registers. EXG is useful for organizing A and B registers properly in the 16-bit D register; for placing information into the more flexible X register; for temporarily swapping stacks; and so forth.



Since I just mentioned the X register as being more flexible, I'll present the command **ABX**. **ABX** instructs the processor to add the value of the B register to the X register. This inherent instruction is very fast, and acts as a kind of fixed increment for X. If X has to move through a high resolution graphics screen hex **$80** bytes at a time, for example, it would be most efficient to set B to **$80** and execute **ABX**. Especially inside a loop, **ABX** would bump the X pointer down to the next graphics screen line in a short time.



Two complementary instructions are add with carry (ADC) and subtract with borrow (SBC). These are standard add and subtract commands, except that the carry/borrow flag is made a part of the computation. I'll talk more about ADC and SBC when I get to the representation of numbers in a later lesson.

TST and BIT are related quick testing instructions. BIT causes the processor to AND the value of an accumulator with a memory location. Certain flags are affected, but the original contents of both accumulator and memory remain unchanged. BIT is particularly useful for locating numbers or ASCII strings in memory, since the value in the accumulator isn't affected as it moves and tests byte after byte.

TST is similar to BIT, but is oriented toward signed numbers. TST tests the value of the operand — which can be a memory location or either accumulator — and sets the negative and zero flags according to what it finds. Signed conditional branches (BGT, BLE, BGE, BLT, BEQ and BNE) are usually placed after the TST.

```
* If the Color Computer's digital-to-analog converter were 8 bits instead of six, what would be the step size (the resolution)?

1/256th of the output.

* What is the step size (the resolution) of the Color Computer's digital-to-analog converter?

1/64th of the output.

* What is the highest resolution of this table of sine values for the Color Computer's digital-to-analog converter?

1/64th of the sine wave shape.

* The following questions refer to the remaining 6809 instructions introduced in Lesson 19.

* What is the action of MUL?

The contents of the A accumulator is multiplied by the contents of the B accumulator, and the result is placed in the D accumulator.

* Is the result of MUL signed or unsigned?

Unsigned.

* If A contains $08 and B contains $C2, what is the result of MUL?

D contains $0610.

* If A contains $55 and B contains $AA, what is the result of MUL?

D contains $3872.

* If A contains $FF and B contains $FF, what is the result of MUL?

D contains $FE01.
```

* What is the result after NOP?

No change to any registers or memory locations; no operation takes place.

* If A contains $08 and B contains $C2, what is the result of EXG A,B?

A contains $C2 and B contains $08.

* If X contains $FFEE and Y contains $01CD, what is the result of EXG X,Y?

X contains $01CD and Y contains $FFEE.

* If X contains $01CD and B contains $33, what is the result of ABX?

X contains **$0200.**

* If X contains $FFFF and B contains $08, what is the result of ABX?

X contains **$0007.**

* If A contains $10 and the carry flag is set, what is the result of ADCA #$10?

$10+$10+C = $21

* If B contains $01, what is the result of SEX?

D contains **$0001.**

* If B contains $FF, what is the result of SEX?

D contains $FFFF.

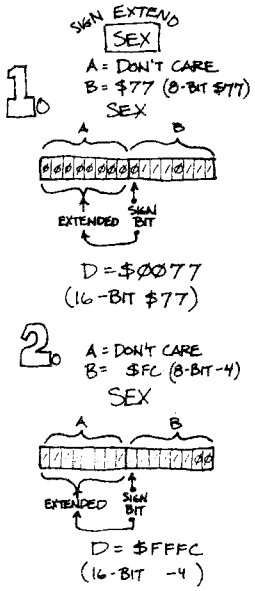
* If B contains $80, what is the result of SEX?

D contains $FF80.

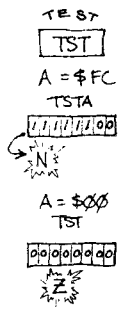
* A contains $43 and ADDA $99 is executed. What is the result after DAA?

A contains $42 and the carry flag is set.

The next instruction also has to do with signed arithmetic. Called sign extend (**SEX**), it results in the sign of the B accumulator being extended into the A accumulator for a complete, signed 16-bit number in the D register. In other words, if B is a positive number, A will become **$00**. If B is **$77**, for example, after **SEX**, the D register will be **$0077**. On the other hand, if B is a negative number, A will become **$FF**. That is, if B is **$FC** (–4 decimal in 8-bit signed arithmetic), a negative number, its sign is extended so that the resulting D register is **$FFFC** — still –4 decimal in 16-bit arithmetic. If that isn't clear, count backwards, first in 8 bits and then in 16 bits. Starting with **$00**, **$FF** is –1, **$FE** is –2, **$FD** is –3, **$FC** is –4. Now start with **$0000**, a 16-bit number. **$FFFF** is –1, **$FFFE** is –2, **$FFFD** is –3, **$FFFC is –4. Sign extend, mnemonic SEX**, sees to it that an 8-bit signed value is properly transformed into a 16-bit signed value.

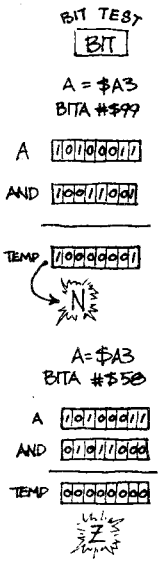


All that's left is **DAA**, the decimal addition adjustment. Microprocessors are working in binary, base 2, and that operation is represented by hexidecimal, base 16. As you've discovered, none of this fits very well with base 10, the decimal system. Some processors contain a decimal mode of operation, where adjustments are made automatically after every computation to compensate for the base 10 system. In other words, no number larger than binary **1001** is allowed in a nybble.



Sadly, decimal mode is is one of the few desirable features not found in the 6809 processor. In its place is the instruction decimal addition adjust, or **DAA**. When executed after and ADD or ADC, the values in the accumulator are converted from true binary mode to a decimal version called binary-coded-decimal, or BCD. The nybbles of the byte are adjusted, and the carry flag set if necessary, to turn the binary result into BCD.

For example, if I were to **LDA #$77** and then **ADDA #$77** (note both these are binary-coded-decimal numbers), the binary result would be hex **#$EE**. Although I want these to be decimal representations, the processor treats them as if they were binary. If I follow those commands with **DAA**, however, a series of tests and corrections are made. **$54** is left in the accumulator and the carry flag is set. That's the number 154 in BCD, the sum of 77 BCD plus 77 BCD. Review the summary of **DAA** on page 43 of your EDTASM+ manual; there will be more on this later.



By the way, it's especially with an operation such as **DAA** that the command ADC comes into play. The carry generated by **DAA** in the previous example has to be taken into consideration when doing arithmetic with larger numbers. Keep that in mind, as I'll be covering that in Representation of Numbers, the next lesson.