22.

What was I saying? Oh yes. Interrupts. Let me take you back to Sam's Kitchen in Roadside, New Jersey, where you can honk for drive up service from noon to 6. Have another listen to Marge at work . . .

Marge: One fries, two BLTs, three chili dogs . . . <honk> Alright, alright . . . and one onion rings. Get those ready. There's a guy out there honkin' that thing like Little Richard. <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> One bunny, two buddies, fries. Where's my order? <at counter> Anything else, Joe? How 'bout you, Mac?

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

Marge: Cheese dog onions.

Kitchen: Orders up.

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

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

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

What was I saying? Oh yes. Interrupts. Having been to Sam's Kitchen twice, you should have an idea that interrupts are crucial to special kinds of programming. But what kind of program would demand such fancy footwork? If the programming is so tricky, why bother?

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

The microprocessor finishes its current instruction, saves important information, and follows programming instructions in response 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.

* Can more than one interrupt occur?

Yes.

* Which interrupt gets taken care of first?

The one with higher priority.

* Can interrupts be ignored?

Yes.

* What permits the processor to ignore 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 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 response to the interrupt.

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

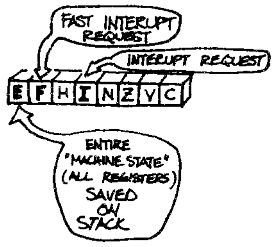
The restaurant is the computer, and Marge is the microprocessor. The cook and customers are program and storage memory. The car horn was the interrupt. Marge finished was she was doing, serviced the interrupt, and returned to finish her previous task. When two interrupts occurred, car two had a higher priority. Finally, the drive-up interrupt was masked out except from noon to six.

The 6809E processor has one power-up reset signal, three hardware and three software interrupts, plus two unique instructions to synchronize itself with hardware interrupts. All of these 6809 interrupts are possible on the Color Computer, and some are already in use by BASIC.

The RESET control is used when the power is turned on to the computer, or when the reset switch is pressed on the back of the machine. It is a separate electrical connection to the 6809 processor, and the RESET cannot be masked by software; it is always accepted.

The most important of the interrupts — that is, the interrupt with the highest priority — is the NMI, or non-maskable interrupt. It is a separate electrical connection to the processor and, like RESET, it cannot be turned off by software. It always commands the attention of the processor.

Of next highest priority is the fast interrupt request, or FIRQ. The FIRQ can be turned off in software by setting bit 6 (the F bit) of the condition code register. **ORCC #$40** can be used to set this bit, turning off the interrupt; **ANDCC #$BF** can be used to clear bit 6 to turn on the interrupt. When the FIRQ comes along, the condition code register and program counter are put on the stack, and the interrupt service routine is begun. The FIRQ is fast because it leaves the remainder of the register stacking up to the interrupt service routine. If a register is not used, it won't need to be put on the stack. I'll talk about the requirements for speed later on.



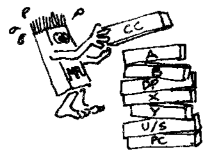
The interrupt with the lowest priority is called simply the interrupt request, or IRQ. When a zero appears on this electrical connection to the CPU, all the registers — what's known as the entire machine state — are saved on the stack. This interrupt is turned off in software by setting bit 4 (the I bit) of the condition codes, and turned on by clearing bit 4. **ORCC #$10** turns it off; **ANDCC #$EF** turns it on.

ORCC #$50 turns off both interrupts; **ANDCC #$AF** turns on both interrupts.

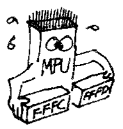
You'll remember that I described indirect addressing by explaining how the computer obtained its first instruction after the power was turned on. The processor went to addresses **$FFFE** and **$FFFF**, concatenated the contents, and used that as the address of the first instruction. There

are in fact seven such address pairs, called "vectors". Power-on reset plus each of the six interrupts has its own vector from **$FFF2** to **$FFFF**.

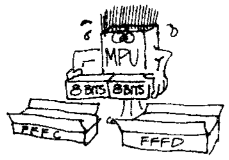
Here's how these vectors look in the Color Computer:



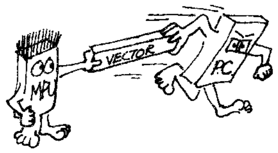
FUNCTION	VECTORS	ADDRESS	CONTENTS
RESET	FFFE+FFFF	AØ27	< BOOT >
NMI	FFFC+FFFD	Ø1Ø9	--------
SWI1	FFFA+FFFB	Ø1Ø6	--------
IRQ	FFF8+FFF9	Ø1ØC	JMP 894C
FIRQ	FFF6+FFF7	Ø1ØF	JMP AØF6
SWI2	FFF4+FFF5	Ø1Ø3	--------
SWI3	FFF2+FFF3	Ø1ØØ	--------



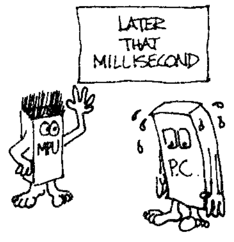
The power-up RESET goes right to address **$A027**, a location in Color BASIC which establishes all the important parameters of the language.



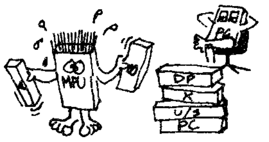
NMI is not used by Color BASIC or Extended Color BASIC, but three unfilled bytes in low RAM are reserved for future use. The future use is provided because the NMI is wired to connection #4 on the computer's cartridge slot.

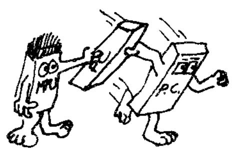


Software interrupts SWI1, SWI2 and SWI3 are also left undefined with three unfilled bytes at their vector locations; they are used by debugging programs such as ZBUG, part of your EDTASM+ cartridge. Yes, we will talk about debugging . . . next time. On to the other interrupts.



FIRQ, the fast interrupt, is hooked to one of the peripheral interface adaptors, connecting to both the PIA's interrupt output lines. The input to the PIA's interrupt control signals are two: the carrier detection (CD) line of the RS-232 communications interface, and the cartridge-in-place (CART) connection, #8 on the computer's cartridge connector. This interrupt serves a dual purpose. When FIRQ occurs, the vector concatenated from addresses **$FFF6** and **$FFF7** point to address **$010F**; at address **010F** is the instruction **JMP $A0F6**, a location in the Color BASIC ROM.





The slower interrupt IRQ is connected to the second peripheral interface adaptor, also to both of its interrupt outputs. The interrupt control inputs of this PIA are connected to the horizontal synchronization (HS) and field or vertical synchronization (FS) outputs of the video display generator. Again, this interrupt serves a dual purpose. When IRQ takes place, the address in vectors **FFF8** and **FFF9** are concatenated to produce address **$010C**. At **$010C** is found the instruction **JMP $894C**, an address in the Extended Color BASIC ROM.

* Is there an interrupt that cannot be masked (turned off)?

Yes.

* What interrupt cannot be masked?

The non-maskable interrupt, or NMI.

* What interrupt has the highest priority?

The NMI.

* What interrupt has the second highest priority?

The fast interrupt request, or FIRQ.

* What bit of the condition code register masks or enables the FIRQ?

Bit 6 masks or enables the FIRQ.

* What information is saved when the FIRQ occurs?

The condition code register and program counter are saved on the stack.

* What is the lowest priority interrupt?

The interrupt request, or IRQ.

* What bit of the condition code register masks or enables the IRQ?

Bit 4 masks or enables the IRQ.

* What information is saved when the IRQ occurs?

All the registers are saved on the stack.

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

Servicing the interrupt.

* How does the program counter find where to go to service the interrupt?

From a vector, or address, in the last 16 bytes of memory.

* What purpose does NMI serve on the Color Computer?

None; it is not used.

* What purpose does FIRQ serve on the Color Computer?

It is used for the RS-232 communications carrier detection line, and for the cartridge-in-place connection on the cartridge connector.

* What purpose does the IRQ serve on the Color Computer?

It is connected to horizontal and vertical synchronization signals from the video display generator.

* What are the terms for vertical and horizontal synchronization with respect to the Color Computer.

Field sync (FS) and horizontal sync (HS).

* How often does the field sync (FS) signal occur?

60 times per second.

* How often does the horizontal sync (HS) signal occur?

15,720 times per second.

* What port address determines which interrupt is fed through to the 6809 processor?

Port address $FF03.

* What condition code bit masks or enables the IRQ?

Bit 4 masks or enables the IRQ.

In all these cases, the addresses in low RAM can be changed or filled in, redirecting the interrupts to any location in memory. You'll be using those addresses.

Now I've given you a formal description of the vectors and the hookup, but I expect it doesn't mean a whole lot to you at this point. I'm going to continue with a detailed description of how everything fits together into a neat package, but first I want you to read the technical information.

Read the MC6809E data booklet page 9 (NMI, FIRQ, IRQ); read the MC6821 data booklet page 7 (peripheral interface lines) and page 8 (internal controls), and Figure 18, page 10; read the MC6847 data booklet page 13 (Field Sync and Horizontal Sync). If you have the Color Computer Technical Reference Manual, read Section III (Theory of Operation). Return to the tape when you have completed the reading.

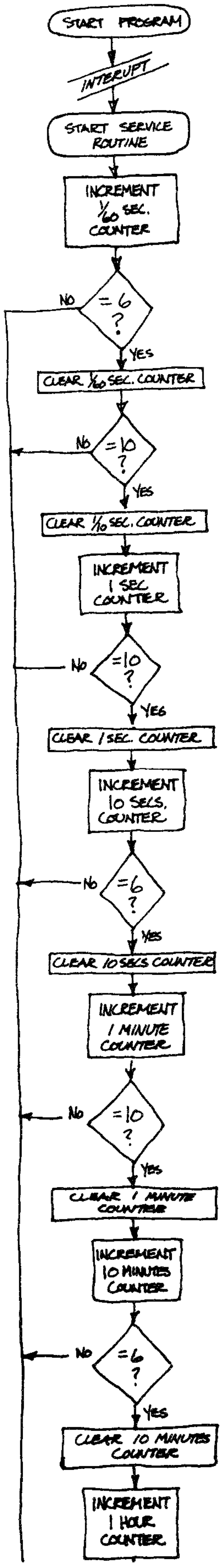
Read the MC6809 data booklet page 9 (NMI, FIRQ, IRQ); read the MC6821 data booklet page 7 (peripheral interface lines) and page 8 (internal controls), and Figure 18, page 10; read the MC6847 data booklet page 13 (Field Sync and Horizontal Sync). If you have the Color Computer Technical Reference Manual, read Section III (Theory of Operation).

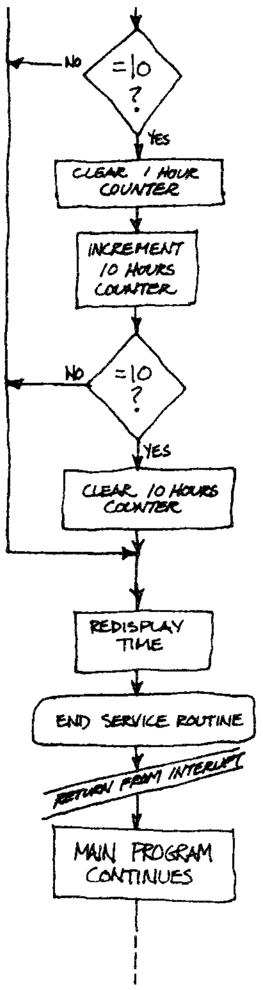
Now putting it together. By correctly writing data to the PIAs, you can make it possible for the computer to detect an RS-232 carrier, to detect the presence of a plug-in cartridge, or to synchronize your programs to the video display either horizontally or vertically. All you need to add is software.

I've got two demonstrations of this. The first is a continuous on-screen software clock; the second, an example of synchronizing the video display with programming changes to the screen.

I'm going to put a clock in the upper right corner of the video screen. It will be there no matter what else is displayed on the screen, whether you're listing, entering or editing a line, or running a BASIC program. It will even keep running with certain machine language programs that don't turn off interrupts or change the vectors. I think I'd like it to count in tenths of a second up to 99 hours, 59 minutes, 59.9 seconds.

You've read the data booklets, so maybe you're ahead of me. Remember the video display generator's field sync (FS) signal, which is used for interrupting the processor. The video display generator's field sync signal occurs at precisely 60 times each second. By enabling the interrupt (bit 0 of port address **$FF03**), I can get an interrupt to occur 60 times each second. If I keep track of those ticks and





update my screen with a new time every six interrupts, then I've got a tenth-of-a-second clock. From a tenth-of-a-second clock I can create a full real-time software clock.

Here's the structure of the setup and interrupt service routine:

1. Set aside some memory for the clock; it might be an image of the actual display (such as 12:59:02.2).

2. Enable the 60-per-second interrupts.

3. On an interrupt, increment the sixtieth-of-a-second counter. If the sixtieth-of-a-second counter passes 5, increment the tenth-of-a-second counter, and clear the sixtieth-of-a-second counter to 0. If the tenth-of-a-second counter passes 9, increment the one-second counter and clear the tenth-of-a-second counter to 0. If the one-second counter passes 9, increment the ten-second counter and clear the one-second counter to 0. If the ten-second counter passes 5, increment the one-minute counter and clear the ten-second counter to 0. If the one-minute counter passes 9, increment the ten-minute counter and clear the one-minute counter to 0. If the ten-minute counter passes 5, increment the one-hour counter and clear the ten-minute counter to 0. If the one-hour counter passes 9, increment the ten-hour counter and clear the one-hour counter to 0. If the ten-hour counter passes 9 clear it to 0.

4. Display the new time to the screen; the re-display will take place every sixtieth of a second, appearing as a continuous display.

5. Clear the interrupt status at port **$FF02**.

6. Return from the interrupt.

The setup process has to clear the way for the interrupts without getting interrupted in the middle of things. So all interrupts go off right at the start; the address of your own routine is placed into the RAM vector; the proper interrupt signal (in this case, the 60-per-second FS) is enabled; interrupts are re-enabled; and the setup routine returns to BASIC. Earlier in the book I presented a map of the computer's input/output port bits. Bit 0 of control port **$FF03** provides for the FS signal to be latched as an interrupt. So the whole routine might look like this:

```
ORCC   #$5Ø      * Turn off interrupts
LDX    #$START   * Service routine start
STX    $Ø1ØD     * Store after "JMP" in vector
LDA    #$37      * Value to enable FS
STA    $FFØ3     * Enable FS through PIA
ANDCC  #$EF      * Re-enable IRQ interrupt
RTS              * Back to BASIC
```

That's the setup. The interrupt service routine itself is really quite simple; get the whole thing loaded into EDTASM+, and then come back for a walk-through.

* What instruction masks the IRQ?

ORCC #$10 masks the IRQ.

* What instruction enables the IRQ?

ANDCC #$EF enables the IRQ.

* What instruction returns to the program in progress after an interrupt has been serviced?

Return from interrupt, RTI.

* When IRQ occurs, where does the program counter obtain the address of the interrupt service routine?

From a vector in high memory.

* What is the IRQ vector found?

The IRQ vector is found at $FFF8 and $FFF9.

* On the Color Computer, where does the IRQ vector point?

The IRQ vector points to address $010C.

* Where is $010C in the Color Computer memory map?

In RAM, on page $01.

* In the Color Computer running with BASIC, the service routine shown in this example ends with JMP $894C. Where is $894C in the Color Computer memory map?

$894C is in the Color BASIC ROM.

* Why does this service routine end with JMP $894C instead of RTI?

Because the interrupt still has to be used by BASIC for the cursor flash and the TIMER command.

Program #34

Program #34, 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.

```
3F00                00100         ORG     $3F00
                    00110 *
3F00 1A    50       00120 INTOFF  ORCC    #$50     * TURN INTERRUPTS OFF
3F02 8E    3F10     00130         LDX     #START   * POINT X TO SERVICE ROUTINE
3F05 BF    010D     00140         STX     $010D    * STORE ROUTINE TO IRQ VECTOR
3F08 86    37       00150         LDA     #$37     * VALUE 00110111 FOR MASKING
3F0A B7    FF03     00160         STA     $FF03    * TURN ON VERTICAL SYNC
3F0D 1C    EF       00170         ANDCC   #$EF     * TURN INTERRUPTS ON
3F0F 39             00180         RTS              * AND BACK TO BASIC "OK"
                    00190 *
3F10 8E    3F77     00200 START   LDX     #IMAGE+10        * POINT X TO 1/10 SEC.
3F13 C6    30       00210         LDB     #$30     * B BECOMES ASCII OFFSET
3F15 6C    84       00220         INC     ,X       * INCREMENT 1/10 SECONDS
3F17 A6    84       00230         LDA     ,X       * GET 1/10 SECONDS VALUE
3F19 81    36       00240         CMPA    #$36     * IS 6/10 SECONDS COUNTED?
3F1B 2D    2C       00250         BLT     OUT      * IF NOT 6/10 SECONDS, OUT
3F1D 8D    40       00260         BSR     DEC1     * ELSE BAC UP 1 MEM. LOCATION
3F1F 81    3A       00270         CMPA    #$3A     * IS IT 1 SECOND YET?
3F21 2D    26       00280         BLT     OUT      * IF NOT 1 SECOND, OUT
3F23 8D    41       00290         BSR     DEC2     * ELSE BACK UP 2 MEM. LOCNS.
3F25 81    3A       00300         CMPA    #$3A     * IS IT 10 SECONDS YET?
3F27 2D    20       00310         BLT     OUT      * IF NOT 10 SECONDS, OUT
3F29 8D    34       00320         BSR     DEC1     * BACK UP 1 MEM. LOCATION
3F2B 81    36       00330         CMPA    #$36     * IS IT 60 SECONDS YET?
3F2D 2D    1A       00340         BLT     OUT      * IF NOT 60 SECONDS, OUT
3F2F 8D    35       00350         BSR     DEC2     * ELSE BACK UP 2 MEM. LOCNS.
3F31 81    3A       00360         CMPA    #$3A     * IS IT 10 MINUTES YET?
3F33 2D    14       00370         BLT     OUT      * IF NOT 10 MINUTES, OUT
3F35 8D    28       00380         BSR     DEC1     * ELSE BACK UP 1 MEM. LOCATION
3F37 81    36       00390         CMPA    #$36     * IS IT 60 MINUTES YET?
3F39 2D    0E       00400         BLT     OUT      * IF NOT 60 MINUTES, OUT
3F3B 8D    29       00410         BSR     DEC2     * ELSE BACK UP 2 MEM. LOCNS.
3F3D 81    3A       00420         CMPA    #$3A     * IS IT 10 HOURS YET?
3F3F 2D    08       00430         BLT     OUT      * IF NOT 10 HOURS, OUT
3F41 8D    1C       00440         BSR     DEC1     * ELSE BACK UP 1 MEM. LOCATION
3F43 81    3A       00450         CMPA    #$3A     * IS IT 100 HOURS YET?
3F45 2D    02       00460         BLT     OUT      * IF NOT 100 HOURS, OUT
3F47 E7    84       00470         STB     ,X       * PLACE $30 (ASCII ZERO)
                    00480 *
3F49 108E  0416     00490 OUT     LDY     #$0416   * POINT TO RIGHT SCREEN
3F4D 8E    3F6D     00500         LDX     #IMAGE   * POINT X TO CLOCK IMAGE
3F50 C6    0A       00510         LDB     #$0A     * COUNT 10 SCREEN POSITIONS
3F52 A6    80       00520 LOOP    LDA     ,X+      * GET CHARACTER FROM CLOCK
3F54 A7    A0       00530         STA     ,Y+      * AND PLACE IT ON THE SCREEN
3F56 5A             00540         DECB             * DONE WITH IMAGE YET?
3F57 26    F9       00550         BNE     LOOP     * IF NOT, THEN GET NEXT CHAR.
                    00560         *
3F59 B6    FF02     00570         LDA     $FF02    * CLEAR VERT. SYNC LATCH
3F5C 7E    894C     00580         JMP     $894C    * AND TO BASIC TO DO RTI
                    00590 *
3F5F E7    84       00600 DEC1    STB     ,X       * PLACE $30 (ASCII ZERO)
3F61 6C    82       00610         INC     ,-X      * BACK UP ONE MEM. LOCATION
3F63 A6    84       00620         LDA     ,X       * GET VALUE FROM IMAGE
3F65 39             00630         RTS              * BACK TO MAIN PROGRAM
                    00640 *
3F66 E7    84       00650 DEC2    STB     ,X       * PLACE $30 (ASCII ZERO)
3F68 6C    83       00660         INC     ,--X     * BACK UP TWO MEM. LOCATIONS
3F6A A6    84       00670         LDA     ,X       * GET VALUE FROM IMAGE
3F6C 39             00680         RTS              * BACK TO MAIN PROGRAM
                    00690 *
3F6D       30       00700 IMAGE   FCC     /00:00:00.00/
           30
           3A
           30
           30
           3A
           30
           30
           2E
           30
           30
                     00710 *
           3F00      00720          END       INTOFF
00000 TOTAL ERRORS
DEC1      3F5F
DEC2      3F66
IMAGE     3F6D
INTOFF    3F00
LOOP      3F52
OUT       3F49
START     3F10
```

The opening is the 16-byte setup routine, turning off interrupts, redirecting the interrupt vector to my interrupt service routine, passing through the 60-per-second interrupt, turning on interrupts, and returning to BASIC.

The service routine itself is a strung-out series of increments and comparisons. The sixtieth-of-a-second clock image in memory is incremented and tested for **$36** (the ASCII value for the character 6). If it's less than six, out it goes; otherwise, it begins a down-the-line test. Notice in the DEC1 and DEC2 routines the use of an indexed pre-decrement command; right along you've been seeing the post-increment commands such as **LDA ,X+**, but this is the first time the pre-decrement has turned up. Since this routine is bumping backwards in memory (from sixtieths of a second up to tens of hours), a decrement is needed.

Check the sequence in the subroutine:

```
STB    ,X
INC    ,-X
LDA    ,X
```

The value in B (an ASCII zero) is stored in memory pointed to by X. The X pointer is decremented and then its contents are incremented. Two things of complementary character are here — the pointer is first decremented, then its contents are incremented. And finally, the A accumulator is loaded with the contents of the memory location now pointed to by X.

After all the increments, tests and updates are complete, the memory image of the time is transferred to the screen. In line 490, Y points to location **$0416** on the screen, and X points to the updated clock. A short loop transfers the information.

Finally, the command **LDA $FF02** resets the latched interrupt from the PIA. In your reading of the MC6821 data booklet, page 8, this was mentioned. I'll read that paragraph. "The four interrupt flag bits are set by active transitions of signals on the four interrupt and peripheral control lines when those lines are programmed to be inputs. These bits cannot be set directly from the MPU data bus and are reset indirectly by a read peripheral data operation on the appropriate section." In other words, flags go up inside the PIA when an interrupt takes place; by reading from the PIA, the flag goes down. **LDA $FF02** reads from the PIA and turns off the interrupt flag.

* What happened to the RTI needed at the end of an interrupt? Where is it?

The RTI is found in the BASIC ROM after it finishes with the cursor flash and timer update.

* When using the MC6821 PIA to cause the interrupt, what is also necessary at the end of the service routine?

The PIA's interrupt latch must be reset.

* What would happen if the latch were not reset?

No further interrupts would pass through the PIA to the processor.

* What two addresses are used by the PIA that handles the IRQ?

Addresses $FF02 and $FF03.

* What command resets the interrupt latch?

Any command that reads from port address $FF02, such as LDA $FF02.

* What does IRQ mean?

IRQ means interrupt request.

* What does PIA mean?

PIA means Peripheral Interface Adapter.

* What do FS and HS mean?

FS means Field Sync and HS means Horizontal Sync.

* What does VDG mean?

VDG means Video Display Generator.

* What does A/IM/AO mean?

Assemble into memory at the absolute origin specified in the source listing.

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

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

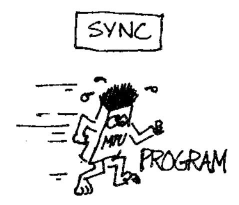
The last instruction (**JMP $894C**) might not make sense to you. You probably expected a return from interrupt instruction (RTI). Let me explain. You'll recall that the interrupt vector for IRQ goes to address **$0110** in low RAM for its instruction. At that location is found the instruction **JMP $894C**. In order for this time display program to work properly with BASIC, it must chain itself to BASIC's vectors. That vector and the subsequent JMP $894C controls the cursor flashing, among other things. So it's go to be there. By replacing **JMP $894C** with the **JMP $3F10** that gets the time display routine going, the program has intercepted a vital part of BASIC's operating system. To keep the link from IRQ vector **$0110** to ROM location **$894C**, this program intercepts **$0110**, patches itself in place, and finishes by jumping to **$894C**. The chain is complete; the time is displayed and BASIC has its cursor. BASIC finishes by executing the return from interrupt (RTI).

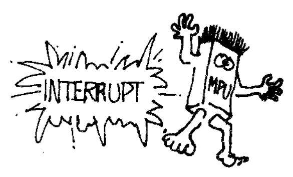
I think it seems simple enough. Give it a try. Assemble this program in memory at the correct origin. Type A/IM/AO and hit enter. The program will assemble into memory. When it's finished and the cursor has returned, type and enter Q. You will quit EDTASM+ and return to BASIC. Protect memory now; this program resides from **$3F00** to **$3F77**, so protect from **$3F00** on up. Type and enter CLEAR200,&H3F00. That's CLEAR200,&H3F00.

Ready? Type and enter EXEC&H3F00.
There's the clock, ticking away in the upper-right-hand corner of the screen. You can enter, edit and list and run BASIC programs. Try a few short programs, and see how it looks to have the clock in the corner.

When you're done with that, try one more test. Create a BASIC program and CSAVE it to tape. I don't care what kind of program it is, and you don't really even need to have the tape running. I just want you to CSAVE something, and keep an eye on the screen. Before the next session, figure out what you see and why it must happen that way. Have fun.

23.





In this lesson, I'm going to turn to video display synchronization achieved with interrupts. But please keep something in mind as you review these past two lessons. This may be the Color Computer you're using, but it's the 6809 processor you're learning to program. Although every 6809 processor is made with these interrupt capabilities and signals, those interrupt signals might be wired in a completely different way on another type of computer. alternative internal wiring might also mean that the vectors in memory would be changed and that the timing of the interrupts would be more or less frequent. Chances are — except for the method used to turn interrupts on and off, which is a function of the 6809's condition code register — everything would be handled differently. Since you're learning the 6809 on the Color Computer, I know these programs will work for you. But if you change computer systems, you'll have to apply the principles but not necessarily the actualities of these interrupt sessions.

That said, it's on to video synchronization. There are only two unique instructions left to talk about on the 6809. These are **SYNC** and **CWAI**.

SYNC and **CWAI** are similar instructions: both cause the 6809 to stop processing — that is, cease to follow program instructions — and wait for an interrupt to occur. **SYNC** (for synchronize) simply turns the processing off, to the point of making it electronically invisible to the rest of the computer components. **SYNC** is especially useful when connecting multiple computers to the same memory; you can't do that with the Color Computer because all the necessary connections aren't there, but **SYNC** makes it possible for some other 6809 computers to work as multiple processor systems.

Like **SYNC**, **CWAI** also causes the processor to stop, but not immediately. **CWAI** (meaning clear condition code bits immediate and wait for interrupt) first places all the registers on the stack and then sets the E flag; the E flag tells the processor that the entire machine state has been

Dealing with interrupts is no more complicated than any assembly programming. The only hitches are getting to the interrupt service routine and back from it without any errors, and, where timing is absolutely critical, getting it over with before it's time for more interrupts. The 60-per-second interrupt in the last lesson's clock program was leisure time at its most relaxing compared with the program in this lesson!

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

Servicing the interrupt.

* How does the program counter find where to go to service the interrupt?

From a vector, or address, in the last 16 bytes of memory.

* What purpose does the IRQ serve on the Color Computer?

It is connected to horizontal and vertical synchronization signals from the video display generator.

Port bits

* What are the terms for vertical and horizontal synchronization with respect to the Color Computer?

Field sync (FS) and horizontal sync (HS).

* How often does the field sync (FS) signal occur?

6Ø times per second.

* How often does the horizontal sync (HS) signal occur?

15,72Ø times per second.

* What port address determines which interrupt is fed through to the 68Ø9 processor?

Port address $FFØ3.

* What condition code bit masks or enables the IRQ?

Bit 4 masks or enables the IRQ.

* What instruction masks the IRQ?

ORCC #$1Ø masks the IRQ.

* What instruction enables the IRQ?

ANDCC #$EF enables the IRQ.

* What instruction returns to the program in progress after an interrupt has been serviced?

Return from interrupt, RTI.

* What is the IRQ vector found?

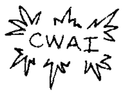
The IRQ vector is found at $FFF8 and $FFF9.

* On the Color Computer, where does the IRQ vector point?

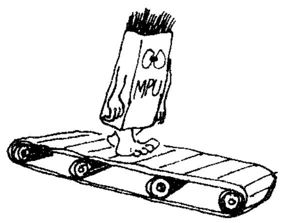
The IRQ vector points to address $Ø1ØC.

saved on the stack. The **CWAI** instruction also keeps the processor active with respect to the outside world; there is no "invisibility" with **CWAI**.

The effective similarity between **SYNC** and **CWAI**, then, is that they both stop the processor's operations and wait for an interrupt to occur. The effective difference is that **SYNC** just stops the operation, whereas **CWAI** also presets the condition codes and saves all the registers.

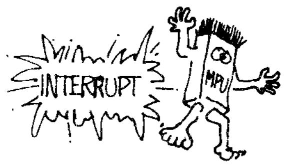


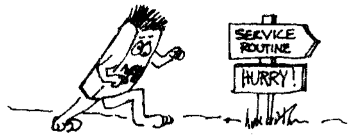
I'll be using **SYNC** for these demonstrations. You might be wondering why stopping the processing with SYNC would be preferable to the straightforward use of an interrupt as I showed you in the last session. With **SYNC**, you can complete all the programming work you need for a change of video contents, then enter **SYNC** mode and wait for further instruction. The amount of time you've got for the program and the timing of the interrupts becomes more important as you write the program, but lets the program work more effectively.

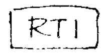


Let me turn back to the peripheral interface adaptors, the PIAs, and their control registers. Addresses **$FF01** and **$FF03** have the important information:

Bit	Function
Ø	Ø = disable interrupt, 1 = enable interrupt request to processor.
1	Ø = falling transition, 1 = rising transition sets IRQA/B1 output.
2	Ø = data direction register, 1 = control register; established at power-up.
3	One of a pair of binary select signals for control of the analog multiplexer (see technical manual for details).
4,5	Establishes CA2/CB2 as output controlled by bit 3 -- always 1 on Color Computer.
6	Interrupt flag when CA2/CB2 is an input; not used on the Color Computer.
7	Interrupt flag from CA1/CB1 -- vertical or horizontal TV synchronization.







Now that you know, what do you do with it? I've got to get technical on you. This is one of those times when hardware meets software, and in order to program what you need, you've got to understand what's going on.

The television screen display isn't a fixed image of some kind, but rather the result of a single, constantly moving electron beam aimed from the back and sweeping across the front of a glass tube. As the beam sweeps by, rare-earth elements known as phosphors are excited by the beam and glow blue, green or red.

By depending on the mixing of the primary colors of blue, green or red (technically called cyan, green and magenta), and also on our eyes' persistence — that is, the ability to retain an image for a small fraction of a second — a complete, multi-colored picture seems to be formed.

If you look at the front of the picture tube with a magnifying glass, you can see the separate colors. By moving your hand quickly in front of the screen, you can see the image "break up" as your hand's outline is strobed by the changing screen image produced by that moving electron beam.

There's only one electron beam, and it's moving fast. It sweeps across the screen, changing color and brightness as it goes, then turns off, sweeps back, turns on, and draws the next line. It draws 262 lines altogether, all the while keeping those lines separated by moving slowly down the screen; one screen full of lines is called a "field". At television speed, "slowly" is only a comparative term, because the beam goes from top to bottom of the screen 60 times each second. On the Color Computer, that's 15,720 lines drawn every second.

What keeps all this happening at the correct time and keeps the beam at the correct place on the screen is known as synchronization. The electrical signal that tells the beam when to start each line across is called horizontal synchronization, or horizontal sync. The signal that tells the beam when to get to the top of the screen and start the next field is called vertical synchronization, or vertical sync. Although it would be simpler to call these horizontal sync and vertical sync, I'm not going to do that. I want to avoid confusing these sync signals with the 6809 processor command **SYNC**.

The MC6847 video display generator, the VDG, creates horizontal and vertical synchronization, and also another signal called field synchronization. Field synchronization is the time between the end of the active display (the very bottom right of the green block that makes up the display screen) and the top of the screen (25 lines before the start of the green block).

For a complete look at all this, open your MC6847 video display generator data booklet, and turn to page 11. On page 11 of the MC6847 data booklet, you can see the relationship between the blank areas and the active display area. Take a few minutes to examine Figures 13 and 14.

---

* Where is $010C in the Color Computer memory map?

In RAM, on page $01.

* When using the MC6821 PIA to cause the interrupt, what is also necessary at the end of the service routine?

The PIA's interrupt latch must be reset.

* What two addresses are used by the PIA that handles the IRQ?

Addresses $FF02 and $FF03.

* What command resets the interrupt latch?

Any command that reads from port address $FF02, such as LDA $FF02.

* What actions does the SYNC instruction cause?

It causes the processor to stop processing instructions and wait for an interrupt to occur.

* What actions does the CWAI instruction cause?

It ANDs the condition code bits with a value, places all the registers on the stack, sets the E flag, stops further processing and waits for an interrupt.

* How are the software actions of SYNC and CWAI alike?

Both stop further processing and wait for an interrupt.

* How are the software actions SYNC and CWAI different?

CWAI (Clear and Wait for Interrupt) performs logical and stack operations, whereas SYNC (Synchronize with Interrupt) does not.

Using FS and HS interrupts

* How are the hardware actions of SYNC and CWAI different?

CWAI keeps the processor active with respect to the outside world (to the other circuits); SYNC makes it electronically invisible (called a tri-state condition).

* How many horizontal lines does the electron beam draw on the video display screen?

262 horizontal lines are drawn on the screen.

* What is one complete group of 262 lines called?

One group of 262 lines comprises a field.

* What is the "green block" in the center of the video screen?

The "green block" is the active display area.

* How many horizontal electron beam lines comprise the active display area?

192 horizontal lines make up the active display area.

* How many fields of 262 lines are drawn each second?

60 fields are drawn each second.

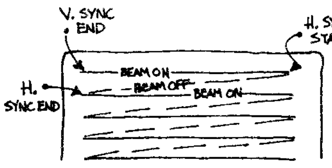
* How many lines are drawn each second?

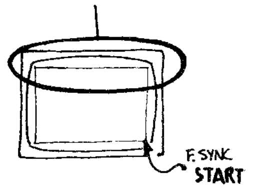
262 lines times 60 fields, or 15,720 lines are drawn each second.

* What controls the horizontal lines and vertical fields?

The Video Display Generator, the VDG.

Turn to page 11 in the MC6847 video display generator (VDG) data booklet and examine Figures 13 and 14, which present the active display area of the Color Computer. Familiarize yourself with the number of horizontal lines and their arrangement. Return to the tape when you have completed the reading.





Don't bite your lip; this is all going to fit together very shortly. When you know about field synchronization and horizontal synchronization, you know two important things. The first thing you know is the time when your processor is free to make its calculations, scan the keyboard, and so forth. That time falls between the end of the active display area and the top of the screen. And that time starts when field synchronization (FS) goes from one to zero, and that time ends when FS goes from zero to one. The 6809 processor can find out when FS changes.

The second thing you know is when the beam starts at the left of the screen and when it ends at the right. It starts when horizontal synchronization (HS) goes from one to zero and ends when HS goes from zero to one. The time when HS is off the screen very short, however (about one CPU clock cycle), so in effect, the important time is the start of the HS period, when HS goes from one to zero. The 6809 processor can find out when HS changes.

So here's an outline of the features as they relate to software.

1. FS goes from high to low. You're out of the screen and free to calculate and perform other operations.

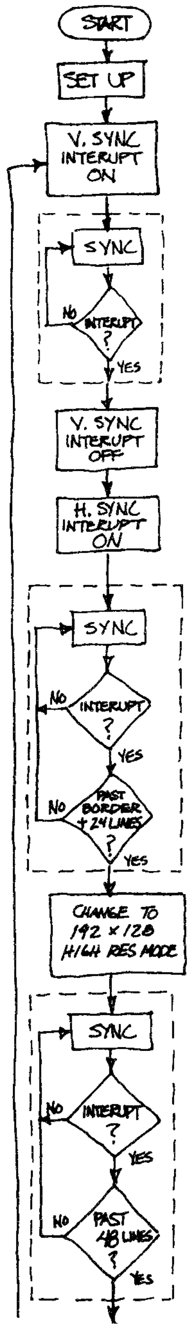
2. FS goes from low to high. You've got to start paying attention to screen lines.

3. HS goes from high to low. The screen has started.

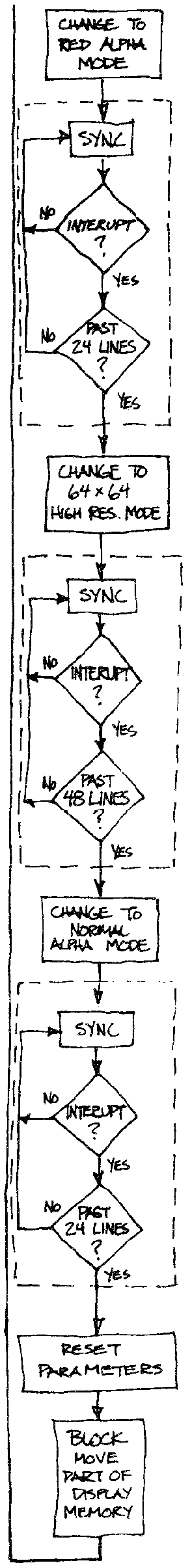
4. Count 38 HS pulses and you're in the display area.

5. 192 HS pulses make up one active screen.

6. Repeat it all 60 times and you've got one full second of programming.



Now it's getting closer. Feed through the vertical or field synchronization to the processor's interrupt, and execute the **SYNC** command. When it occurs, execute a vertical synchronization service routine. That routine should turn off that feed-through and turn on the horizontal synchronization feed-through. Create another interrupt service routine for the horizontal synchronization. Begin



counting until you reach the top of the active display area. Then you can change the display and count screen lines in short programming bursts, ending each with **SYNC**. When you have counted 192 lines, the screen display area is completed. You can turn off the horizontal feed-through, turn back on the vertical synchronization feed-through, return to the main loop for your calculations and more sophisticated programming. When that's done, you can execute the **SYNC** command and wait for the process to start all over.

A practical example is the only way of understanding what this is good for and how to use it. Before that, though, please review this lesson so far, reread the control register information in the MC6821 peripheral interface adapator data booklet, and re-examine the screen outline on page 11 in the MC6847 data booklet.

Review this lesson. After reviewing, read the control register information in the MC6821 data booklet, pages 7 and 8. Also continue to become familiar with the screen outlines on page 11 of the MC6847 VDG data booklet. Return to the tape when you have completed the reading.

The practical example I've got is about as impractical as they come in some respects. It shows a bunch of random colors and shapes on the screen, together with alphanumerics. There are standard letters and characters (black on green), high resolution color graphics, more characters (black on red), medium resolution color graphics, and more characters. The trick is that all of them are displayed on the same screen at the same time.

Getting a mix of high-resolution graphics and standard alphanumerics on the screen at the same time is a simple function of synchronizing and counting. If you synchronize to the vertical synchronization pulse, you know where the screen starts. If you synchronize to the horizontal synchronization pulse, you know where each of the 192 screen lines is. If you are familiar with your graphics modes, then you know what character is where on what line.

All that's left is the implementation. My example presents two rows of alphanumeric characters, a 192 by 48 block of high resolution color graphics, two more rows of alpha characters (but in red instead of green), a 64 by 16 block of medium resolution color graphics, and three rows of alpha characters. I haven't filled memory with anything in particular, so it's just random junk. But the junk'll be moving. Load the source code. I'll take you through it, and do some explaining.

* What is FS (Field Synchronization) on the VDG?

The time between the end of the active display area and the top of the screen.

* When does FS go from high to low (one to zero)?

When the electron beam leaves the active display area.

* When does FS go from low to high (zero to one)?

When the electron beam reaches the top of the screen.

* When does HS go from one to zero?

When the electron beam begins drawing a line on the screen.

* When does HS go from zero to one?

When the electon beam finishes drawing a line on the screen.

* According to the MC6847 data booklet, how many HS pulses occur before the "green block" -- the active display area -- begins?

38 HS pulses occur before the active display area begins.

* How many HS pulses occur during active display (within the "green block")?

192 HS pulses occur within the active display.

* According to the MC6847 data booklet, how many HS pulses occur after the active display area ends?

32 HS pulses occur after the active display area ends.

Program #35, an EDTASM+ program. Insert the EDTASM+ cartridge, and turn on the power to your computer. When the cursor appears, typle 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/0 error occurs, rewind to the program's start and try again. For severe loading problems, see the Appendix.

```
          000C      00100 ROW      EQU      12
          0023      00110 BORDER   EQU      35
          0420      00120 HIRES    EQU      $0420
          0800      00130 VIDTOP   EQU      $0800
          FF00      00140 CLEARH   EQU      $FF00
          FF02      00150 CLEARV   EQU      $FF02
          FF01      00160 HSPORT   EQU      $FF01
          FF03      00170 VSPORT   EQU      $FF03
          010D      00180 VECTOR   EQU      $010D
          FFC0      00190 VIDCL0   EQU      $FFC0
          FFC1      00200 VIDST0   EQU      $FFC1
          FFC2      00210 VIDCL1   EQU      $FFC2
          FFC3      00220 VIDST1   EQU      $FFC3
          FFC4      00230 VIDCL2   EQU      $FFC4
          FFC5      00240 VIDST2   EQU      $FFC5
          FF22      00250 VIDPRT   EQU      $FF22
                    00260 *
3F00                00270          ORG      $3F00
                    00280 *
                    00290 * GET & SAVE BASIC VECTOR
                    00300 * PLACE THIS VECTOR
3F00 1A   50        00310 BEGIN    ORCC     #$50
3F02 BE   010D      00320          LDX      VECTOR
3F05 BF   3FC8      00330          STX      STOREV
3F08 8E   3F7D      00340          LDX      #INTER
3F0B BF   010D      00350          STX      VECTOR
                    00360 *
                    00370 * INTERRUPTS OFF.
                    00380 * HORIZONTAL SYNC OFF.
                    00390 * VERTICAL SYNC ON.
                    00400 * SELECT ALPHA MODE.
                    00410 * INTERRUPTS ON.
                    00420 * WAIT FOR VERTICAL SYNC.
3F0E 86   36        00430 STAR     LDA      #$36
3F10 B7   FF01      00440          STA      HSPORT
3F13 4C             00450          INCA
3F14 B7   FF03      00460          STA      VSPORT
3F17 B7   FFC4      00470          STA      VIDCL2
3F1A B7   FFC2      00480          STA      VIDCL1
3F1D B7   FFC0      00490          STA      VIDCL0
3F20 8E   3F7F      00500          LDX      #SCREEN
3F23 1C   EF        00510          ANDCC    #$EF
3F25 13             00520          SYNC
                    00530 *
                    00540 * WAIT FOR HORIZ. SYNC.
                    00550 * COUNT BORDR + 24 LINES.
                    00560 * CHANGE TO 128X192 COLOR
3F26 8E   3F94      00570          LDX      #LINE
3F29 C6   3B        00580          LDB      #BORDER+2*ROW
3F2B 1C   EF        00590          ANDCC    #$EF
3F2D 13             00600 LOOP1    SYNC
3F2E 5A             00610          DECB
3F2F 26   FC        00620          BNE      LOOP1
3F31 86   EF        00630          LDA      #$EF
3F33 B7   FF22      00640          STA      VIDPRT
3F36 B7   FFC5      00650          STA      VIDST2
3F39 B7   FFC3      00660          STA      VIDST1
                    00670 *
                    00680 * WAIT FOR HORIZ. SYNC.
                    00690 * COUNT 48 LINES.
                    00700 * CHANGE TO ALPHA MODE.
3F3C C6   30        00710          LDB      #4*ROW
3F3E 13             00720 LOOP2    SYNC
3F3F 5A             00730          DECB
3F40 26   FC        00740          BNE      LOOP2
3F42 86   0F        00750          LDA      #$0F
3F44 B7   FF22      00760          STA      VIDPRT
3F47 B7   FFC4      00770          STA      VIDCL2
3F4A B7   FFC2      00780          STA      VIDCL1
```

```
                  *
                  * WAIT FOR HORIZ. SYNC.
                  * COUNT 24 LINES.
                  * CHANGE TO 64X64 COLOR.
3F4D C6    18               LDB     #2*ROW
3F4F 13           LOOP3     SYNC
3F50 5A                     DECB
3F51 26    FC               BNE     LOOP3
3F53 86    8F               LDA     #$8F
3F55 B7    FF22             STA     VIDPRT
3F58 B7    FFC4             STA     VIDCL2
3F5B B7    FFC2             STA     VIDCL1
3F5E B7    FFC1             STA     VIDST0
                  *
                  * WAIT FOR HORIZ. SYNC.
                  * COUNT 48 LINES.
                  * CHANGE TO ALPHA MODE.
3F61 C6    30               LDB     #4*ROW
3F63 13           LOOP4     SYNC
3F64 5A                     DECB
3F65 26    FC               BNE     LOOP4
3F67 86    07               LDA     #$07
3F69 B7    FF22             STA     VIDPRT
3F6C B7    FFC0             STA     VIDCL0
                  *
                  * WAIT FOR HORIZ. SYNC.
                  * COUNT 48 LINES.
3F6F C6    30               LDB     #4*ROW
3F71 13           LOOP5     SYNC
3F72 5A                     DECB
3F73 26    FC               BNE     LOOP5
                  *
                  * INTERRUPTS OFF.
                  * DO BYTE FINAGLE STUFF.
                  * START IT ALL AGAIN.
3F75 1A    50     STOP      ORCC    #$50
3F77 BD    3F98             JSR     INCREM
3F7A 7E    3F0E             JMP     STAR
                  *
                  * SUBROUTINES FOLLOW.
                  * JUMP OFFSET INDEXED.
                  * X POINTS TO ROUTINE.
3F7D 6E    84     INTER     JMP     ,X
                  *
                  * CLEAR FIELD SYNC LATCH.
                  * SELECT ALPHA MODE.
                  * TURN VERTICAL SYNC OFF.
                  * TURN HORIZ. SYNC ON.
                  * CLEAR HOR. SYNC LATCH.
                  * BACK TO MAIN PROGRAM.
3F7F B6    FF02   SCREEN    LDA     CLEARV
3F82 86    07               LDA     #$07
3F84 B7    FF22             STA     VIDPRT
3F87 86    36               LDA     #$36
3F89 B7    FF03             STA     VSPORT
3F8C 4C                     INCA
3F8D B7    FF01             STA     HSPORT
3F90 B6    FF00             LDA     CLEARH
3F93 3B                     RTI
                  *
                  * CLEAR HOR. SYNC LATCH.
                  * BACK TO MAIN PROGRAM.
3F94 B6    FF00   LINE      LDA     CLEARH
3F97 3B                     RTI
                  *
                  * BYTE-FINAGLE ROUTINE.
                  * BLOCK MOVES $44 BYTES
                  *  AT A TIME, CONTINUING
                  *  UNTIL #VIDTOP IS
                  *  REACHED.
                  * RESETS STORAGE AND
                  *  START LOCATIONS,
                  *  INCREMENTS Y TO NEXT
                  *  BLOCK MOVE POINT.
3F98 BE    3FC2   INCREM    LDX     XSTORE
3F9B 10BE  3FC4             LDY     YSTORE
3F9F C6    44               LDB     #$44
3FA1 A6    A0     FILLUP    LDA     ,Y+
3FA3 A7    80               STA     ,X+
3FA5 5A                     DECB
3FA6 26    F9               BNE     FILLUP
3FA8 8C    0800             CMPX    #VIDTOP
3FAB 2D    0D               BLT     VIDMOR
```

```
3FAD 8E   0420                  LDX    #HIRES
3FB0 10BE 3FC6                  LDY    YHOLD
3FB4 31   21                    LEAY   1,Y
3FB6 10BF 3FC6                  STY    YHOLD
3FBA 10BF 3FC4         VIDMOR   STY    YSTORE
3FBE BF   3FC2                  STX    XSTORE
3FC1 39                         RTS
                       *
3FC2      0600         XSTORE   FDB    $0600
3FC4      0000         YSTORE   FDB    $0000
3FC6      0000         YHOLD    FDB    $0000
3FC8                   STOREV   RMB    02
                       *
          3FCA         ZZZZZZ   EQU    *
                       *
          3F00                  END    BEGIN
00000 TOTAL ERRORS
BEGIN   3F00
BORDER  0023
CLEARH  FF00
CLEARV  FF02
FILLUP  3FA1
HIRES   0420
HSPORT  FF01
INCREM  3F98
INTER   3F7D
LINE    3F94
LOOP1   3F2D
LOOP2   3F3E
LOOP3   3F4F
LOOP4   3F63
LOOP5   3F71
ROW     000C
SCREEN  3F7F
STAR    3F0E
STOP    3F75
STOREV  3FC8
VECTOR  010D
VIDCL0  FFC0
VIDCL1  FFC2
VIDCL2  FFC4
VIDMOR  3FBA
VIDPRT  FF22
VIDST0  FFC1
VIDST1  FFC3
VIDST2  FFC5
VIDTOP  0800
VSPORT  FF03
XSTORE  3FC2
YHOLD   3FC6
YSTORE  3FC4
ZZZZZZ  3FCA
```

I've prepared this source listing to make full use of labels. Print the first screenful of lines; start with me at the top.

* What happens at the end of the active display area?

FS goes from high to low (one to zero).

Internally, the MC6847 video display generator counts to 12, which is the number of horizontal lines that make up a single alpha character position; so I label 12 as ROW. The upper border is defined by the 6847, so I label that BORDER. I'll be moving some display bytes around for effect; these moving display bytes will start at memory labeled HIRES and end at memory labeled VIDTOP.

* What PIA address handles the FS interrupt?

Port address $FF03.

* What PIA address resets the FS interrupt?

Reading port address $FF02.

The remaining are labels of key function addresses in upper memory; some you've seen before. As you have read in the MC6821 data booklet, the horizontal synchronization interrupt is cleared by reading **$FF00** and the vertical synchronization interrupt is cleared by reading **$FF02**; they are labeled CLEARH and CLEARV. The actual synchronization interrupts are fed through to the 6809's IRQ line by writing enabling information to ports **$FF01** and **$FF03**, here labeled HSPORT and VSPORT.

* What PIA address handles the HS interrupt?

Port address $FF01.

The IRQ vector from high memory finds its commands as the operand of the JMP at **$010C**, so **$010D** is labeled VECTOR.

There are six SAM addresses that control the video modes. The odd addresses clear the mode bit to zero, the even addresses set the bit to one. You know that. So mode bits 0, 1 and 2 are labeled VIDCL0 and VIDST0, VIDCL1 and VIDST1, VIDCL2 and VIDST2. Finally, the port address for the remaining video controls is found at **$FF22**; it's labeled VIDPRT.

Now display the last few lines of the program; begin at line 1500. P1500:*. Labels XSTORE, YSTORE and STOREV are two-byte groups set aside for temporary storage of video positions between vertical synchronization pulses.

So now you know the pack of labels I've got here. I've tried not to clutter this listing with lots of comments, so follow with me now. The first block of code turns off all interrupts which may have been enabled, and replaces the IRQ vector at **$010D** in RAM with my interrupt service routine. In the next block, horizontal synchronization interrupts are turned off, vertical synchronization interrupts are turned on, and alphanumeric video mode is selected.

The X register is loaded with a pointer to the vertical synchronization service routine, interrupts are enabled, and the processor enters **SYNC** mode. It now waits for the vertical synchronization pulse to force an interrupt. When the vertical synchronization interrupt occurs, the interrupt service routine is entered.

This routine finds the proper service by performing a zero-offset indexed jump based on the contents of the X register. Since X was pointed to the routine labeled SCREEN, this routine is performed. The SCREEN service routine clears the vertical synchronization latch, selects alpha mode, turns off the vertical synchronization feed-through, turns of the horizontal synchronization feed-through, clears the horizontal synchronization latch, and returns from the interrupt. It returns with everything set up for being interrupted by the horizontal synchronization pulse.

In other words, when the program starts, everything sets up and waits for the SCREEN service routine, which identifies the top of the screen and sets things up for the 262 horizontal interrupts.

The return from interrupt brings things back in the program to where the X register is pointed to the LINE service routine, the B register is set up to count through the screen border lines and 24 displayed lines. Remember I'm talking about electron beam lines here, not the usual lines of text. Interrupts are enabled, and the **SYNC** wait is on.

* What PIA address resets the HS interrupt?

Reading port address $FF00.

* What two items control the VDG modes?

Port $FF22 and the SAM control the various VDG modes.

* What is the general term for setting up the PIA or the VDG?

Configuring.

* After configuring the PIA for interrupts and the VDG for modes, the address of the interrupt service routine is put in place. How is that address accessed?

Through the IRQ vector in high memory.

* Where is the IRQ vector in high memory, and where does it point on the Color Computer?

The IRQ vector is at $FFF8 and $FFF9, and points to $010C in RAM.

* What addressing mode is this?

Indirect addressing.

* What does IRQ mean?

Interrupt request.

* How often does the horizontal interrupt HS occur?

15,720 times per second.

* According to the MC6847 data booklet, about how long is this?

It is approximately 63.5 microseconds.

* How many 6809 clock cycles is this on the Color Computer?

63.5 divided by 1.11746 is under 57 clock cycles.

* What actions does the SYNC instruction cause?

It causes the processor to stop processing instructions and wait for an interrupt to occur.

* What actions does the CWAI instruction cause?

It ANDs the conditon code bits with a value, places all the registers on the stack, sets the E flag, stops further processing and waits for an interrupt.

* How are the software actions of SYNC and CWAI alike?

Both stop further processing and wait for an interrupt.

* How are the software actions SYNC and CWAI different?

CWAI (Clear and Wait for Interrupt) performs logical and stack operations, whereas SYNC (Synchronize with Interrupt) does not.

* How many horizontal lines does the electron beam draw on the video display screen?

262 horizontal lines are drawn on the screen.

* What is one complete group of 262 lines called?

One group of 262 lines comprises a field.

* What is the "green block" in the center of the video screen?

The "green block" is the active display area.

The LINE service routine, arrived at through the zero-offset-indexed jump, merely clears the horizontal interrupt and returns. The B register is decremented, and if the selected number of electron beam lines is not yet counted through, **SYNC** is entered again. When the count is finished, the video mode is changed, the row counter recharged with a new value, and the **SYNC** state re-established.

There are five of these horizontal **SYNC** loops, each changing the video mode after a specific number of horizontal lines have been completed.

After the top border plus 192 horizontal lines, the active display area is complete and interrupts are disabled by the program. A short byte-move subroutine is called — you can put anything you like here — which bumps some display bytes around in the high resolution area. It lets you know something is happening. After the return from that byte-finagling subroutine, the process of vertical and horizontal synchronization starts again.

There are some important things to know. First of all, the horizontal interrupt occurs about ever 63.5 microseconds. That means you've got just about 57 clock cycles to perform your horizontal interrupt service routine. LOOP3 is the longest — I'll leave the calculations to you — but it makes it.

The other critical timing depends on the value of B (**$44** in my example) used to count bytes moved between vertical synchronization interrupts. In this case, **$44** is the highest number of moves I could fit between pulses.

Now keep in mind that this is a relatively crude demonstration of the possibilities of video manipulation. If you're interested in creating fast games or using powerful graphics capabilities, this method should give you as much power as any of the famous commercial game machines.

Now try it. Assemble this in memory by typing A/IM/AO/NL/NS. Assemble in memory at the absolute origin with no listing and no symbol table displayed. That's A/IM/AO/NL/NS. In a few seconds, the prompt and cursor will reture. Quit the editor/assembler by typing and entering Q. When the BASIC sign on message appears, you're ready for the demo. Type and enter EXEC&H3F00. That's where it all starts. EXEC&H3F00.

There's your mixed-mode display with moving parts. Study the listing and review this lesson; next time the trials and tribulations of debugging, hints and ideas, and a summary of what you have been learning. Till then.

24.

Welcome back. Up to this point, you've been walking an unfamiliar but well-lit path through assembly language. When this road ends, though, you'll be staring ahead into a kind of wilderness. If you know the natural signs, the footprints in the snow, how to feed and shelter yourself, then you'll survive to create your own paths. This course has been your outdoor survival training.

But that country isn't like this city, so you'll need not only the kit of tools — the editor/assembler, the data booklets, and the knowledge — but you'll also need something to cut a path in the underbrush so you can see through the woods and ahead to your destination.

That tool is a debugger. Sometimes called a machine-language monitor, the debugger is a program which displays memory contents, takes memory contents apart and translates them into mnemonics, does calculations and even steps through programs an instruction at a time.

The debugger is the "plus" in EDTASM+. This debugger is called ZBUG; get it ready now. Turn off your computer, insert the EDTASM+ cartridge, and turn the computer back on. The usual star prompt and flashing EDTASM cursor will appear. Type Z and press ENTER. The star prompt has changed to a crosshatch. You are in the ZBUG monitor. Now type E and press ENTER. Your star prompt returns and you are back in EDTASM.

Start with a program; you'll be doing the typing in this final lesson. The program is shown in the book. Enter it with the usual EDTASM insert-line mode (I), and assemble it to memory at the origin shown (A/IM/AO):

```
        ORG     $3FØØ
VIDEO   EQU     $Ø48Ø
COUNT   EQU     $ØØØØ
START   LDX     #VIDEO
        LDB     #COUNT
        LDA     #$FF
LOOP    INCA
```

As I come to the end of this course, it feels to me like a great novel should be ending, with its stereotypical sunsets, tears or flourishes. Rather than that, it's just some debugging and summaries. Maybe later for my Great American Novel; for now, you finish learning the 6809.

* What is another name for a debugging program?

A machine language monitor.

* What is the name of the machine language monitor that is part of EDTASM+?

ZBUG is the debugger.

* What is a breakpoint?

A stopping place in a machine language program inserted for debugging purposes.

* What is used as a breakpoint in ZBUG?

The software interrupt SWI.

```
STA     ,X+
DECB
BNE     LOOP
SWI
END     START
```

When it's assembled, enter ZBUG by typing Z and pressing ENTER. Have a look at the assembly; your origin was **$3F00**, so type **3F00** followed by a slash. **3F00/** reveals **LDX #VIDEO**. When you do an in-memory assembly, ZBUG references your labels. Start pressing the down arrow. The program is being shown to you command by command, with each labeled as in the program.

Type BREAK. Again type **3F00**, but this time follow it with a comma. Type a few more commas, and continue tapping the comma quickly. Watch the screen carefully as you scroll through the commands. Reverse-video characters begin to appear and scroll up the screen. Eventually these change to normal characters, and finally to graphics characters. The program is executing step by step; the instructions . . .

```
LOOP  INCA
      STA  ,X+
      DECB
      BNE  LOOP
```

. . . are passing by and actually performing their functions. Keep pressing the comma. It takes four taps of the comma to produce one character, so you can see the repetitive nature of this program.

The character of the program is already familiar to you. It's nothing more than a memory fill starting at **$0480** and continuing for 256 loop repetitions.

Now watch it work at full speed; go to the start. Type G3F00 and tap ENTER. G3F00 ENTER. The screen gets blasted instantaneously with 256 characters, and ZBUG prints "8 BRK @ LOOP+6". There's the software interrupt command at work. Don't remember it? Tap E and return to EDTASM, and print the source code (P#:*) on the screen.

Right before the END statement is SWI, the software interrupt. This is what ZBUG uses for its breakpoints. More about breakpoints in a minute; back to ZBUG. Type and enter Z.

You've seen the labels in this listing. Now look at the actual hex values. Type H and hit ENTER. Now type **3F00/** and examine the display. Instead of symbolic notation using the labels (it used to read **LDX #VIDEO**) you now see **LDX #480**. Oh yes. The default notation in ZBUG is hexadecimal.

Type S and hit ENTER. Now 3F00/ once again reveals **LDX #VIDEO**. Another of these. Type B and ENTER. 3F00/ now shows **8E**, the hexadecimal value at memory

* What happens when a program encounters a software interrupt?

All the registers are saved and the program counter obtains the SWI vector from high memory.

* What is another name for "all the registers"?

The machine state.

* When an interrupt saves the machine state, what flag does it set?

The E, or entire state, flag.

* What is another name for a machine language monitor?

A debugger.

* The following questions summarize the concepts you should have learned from this course.

* How are machine language impulses represented?

By ones are zeros.

* What numbers system consists only of ones and zeros?

The binary system.

* What is the abbreviation for binary digit, and what is a group of four and a group of eight binary digits called?

A binary digit is a bit; four binary digits is a nybble (nibble); eight binary digits is a byte.

* What number system is used in programming for the convenient representation of binary numbers?

The hexadecimal number system.

location **$3F00**. Hitting the down arrow reveals labeled locations with hexadecimal values. And finally, one more to try. Type and enter A. Tap the down arrow and you see ASCII characters.

There's some preliminary work with ZBUG. Now you have reading to do. Chapters 2, 5 and 6 of the EDTASM+ manual have a complete description of the features of ZBUG. Read all the chapters and try the examples presented in the manual. Pay particular attention to the use of breakpoints — stopping places in the program — and the ways you can examine and change both memory and register contents. This powerful debugger will make finding those program glitches and endless loops lots easier. When you're done with the reading, come back for some suggestions on using ZBUG, and for a final summary of this course.

Using ZBUG is time consuming, but worthwhile. You might get tired of going through a long delay loop, though. In a case like this, use the register examination mode to change the loop value so it's almost done. Then you can continue execution, and the loop will complete.

One type of program that is almost impossible to debug in this manner is the interrupt-driven program. Enabling and disabling interrupts can be done, but when it comes to their actual execution, ZBUG will hang up, waiting for the interrupts which will never come. So for this kind of program, try your debugging by changing interrupts to subroutines in key places, saving and restoring the entire machine state (all the registers), and simulating the interrupts.

With the ability to use multiple number systems, to provide automatic calculations, to single-step your programs, and to display memory and registers, ZBUG is your most important tool — other than your own careful thinking and programming — in completing working, speedy and efficient assembly language programs. At the end of this lesson, use ZBUG to examine and execute each of the assembly language programs in this course.

In this course you have learned that assembly language is a representation of machine language, a carefully organized pattern of electronic impulses. These electronic impulses directly manipulate the actions of the microprocessor, and are therefore extremely fast and can be organized to perform any function which the computer's hardware permits. As patterns of electronic impulses, this kind of programming is distinctly different from high-level languages such as BASIC, languages which are in themselves constructed from large-scale patterns of machine commands.

Machine language consists of electronic impulses which are best expressed as one and zero conditions. The binary

* What does ASCII mean? What is it used for?

ASCII means American Standard Code for Information Interchange; ASCII is a binary pattern of control codes and characters used for computer communication and display.

* What is the overall organization of a processor called?

The architecture.

* Describe the architecture of the 6809 processor.

The 6809 consists of an Arithmetic Logic Unit and an Instruction Decoder; a program counter PC, accumulators A and B, index registers X and Y, stack pointers S and U, direct page register DP, and condition code register CC; A and B can be combined into accumulator D.

* What are processor commands called? What is the data used by the commands called?

Processor commands are operation codes, or opcodes; the data used by the commands are the operands.

* What are the verbal descriptions of processor commands called? What is a program listing containing these descriptions called?

Verbal descriptions are mnemonics, and a program listing containing mnemonics is called source code.

* What does an assembler do?

An assembler translates source code into object, or binary, code.

* What is an addressing mode?

An addressing mode is the way a machine language program gets the information it needs to complete an instruction.

* What are the 6809's addressing modes?

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

* Describe inherent addressing.

The mode in which the opcode contains all the information the processor needs to complete the instruction.

* Describe register addressing.

The mode in which a postbyte describes the registers which are used to complete the instruction.

* Describe immediate addressing.

The mode in which the information to complete the instruction immediately follows the opcode.

* Describe extended addressing.

The mode in which the information is found at the address given after the opcode.

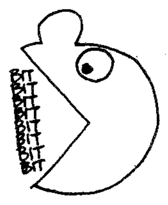
* Describe direct addressing.

The mode in which the information is found at the address calculated from the direct page register and the value following the opcode.

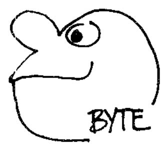
* Describe indexed addressing.

The mode in which the information is found at the address calculated from a fixed or variable offset and an index register.

system is a representation of ones and zeros, so the binary system counts in powers of two. The binary digits (the bits) are organized in groups of eight. These eight-bit groups are called bytes, and the byte is the word size for the 6809 processor.



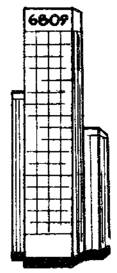
6809 words can stand for commands, data, characters, and can be used for counting and distances. When 6809 words are used as characters, those words are patterned in accordance with the American Standard Code for Information Interchange (ASCII).



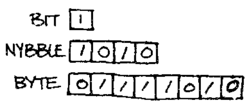
All microprocessors have an overall organization known as architecture. The architecture of the 6809 encompasses its internal architecture, plus the ability to address 65,536 bytes of external memory. The internal architecture includes an arithmetic logic unit (ALU), an instruction decoder (ID), a 16-bit program counter (PC), two 8-bit accumulators (A and B), two 16-bit index registers (X and Y), two 16-bit stack points (S and U), an 8-bit direct page register (DP), and an 8-bit condition code register holding the flags (CC). The two 8-bit accumulators A and B can be combined to produce the 16-bit accumulator D.

$5A = ASCII Z
$5A = command DEC B

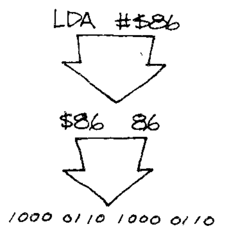
Commands to the 6809 processor are electronic impulses, represented by binary digits, and organized as bytes. The binary bytes are themselves thought of as two 4-bit groups, each of which is represented in hexadecimal notation. Hexadecimal notation, also called hex, counts from **0** through **F** and best expresses the character of 4-bit group. The 4-bit half of a byte is sometimes called a nybble.



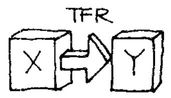
The hexadecimal notation represents the binary patterns, but the commands themselves are further abstracted into verbal descriptions. The verbal descriptions are called mnemonics, and the mnemonics are used for the construction of source code. Source code is a readable, quasi-verbal description of the processor actions that make up a complete program.

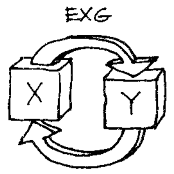


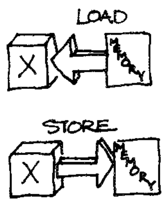
Source code is made up of mnemonics for binary machine commands, called opcodes, and the necessary information to complete the command, called the operand. Opcodes and operands — together with labels, origins, ends, byte descriptions, comments, and other information — make up the complete source listing. The source listing is entered and edited using an assembler, and translated from its source form to machine language by an assembler. The assembler takes the source code and produces from it the machine language, called object code.



The most common machine instructions move information inside the processor, move information from the processor to memory, and from the memory to the processor. These are transfers, exchanges, stores and loads. The processor manipulates this information through arithmetic and logical functions. The arithmetic includes addition, subtraction, multiplication, incrementing and decrementing. The logic includes AND, OR, Complement, Negation,







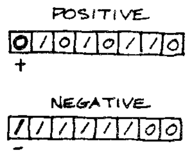
Exclusive-OR. Other processor manipulations of data include shifting or rotating bits left or right, testing for bits, comparison with other data, clearing to zero, and special functions for decimal addition and positive and negative arithmetic.

The processor obtains its information by providing the address of the data in external memory. The processor can determine the address it needs in a variety of simple and complex ways. These techniques are called addressing modes.

Among the addressing modes in the 6809 processor are inherent, register, immediate, extended, direct and indexed. The inherent mode contains all the information the processor needs to complete an instruction. The register mode specifies information which informs the processor what internal registers to use. The immediate mode provides the processor with a value to use directly. The extended mode gives the processor an address at which it can find the information it needs. The direct mode combines the special direct page register with information to locate the data in memory. The indexed mode calculates a result from register information and fixed or variable offsets, and uses the results of that calculation to find the data in memory. Automatic incrementing or decrementing of certain registers can be specified in the indexed addressing mode. The relative mode instructs the processor to find information in relation to its current position in memory.

INHERENT
CLRA
REGISTER
TFR X,Y
IMMEDIATE
LDX #$0400
EXTENDED
LDY $1234
DIRECT
LDX <$33
INDEXED
LDB $41,X
INDIRECT
LDA [$19,Y]

One of the features of the 6809 processor which speeds its operation and makes access of data simpler is the indexed indirect addressing mode. This mode applies to most of the previous indexing modes, and permits the processor to access information through a second level. The data is found at the address specified by the data found at an address determined by the processor from the instruction of the operand. This doesn't lend itself to a summary, so refer to lessons 15, 16 and 17 for more.



Great program structure is achieved using the indexed indirect addressing mode. By using an index relative to the current position of the program counter, complete program position independence within memory can be achieved.

The information actually received by the processor through all these adddressing modes is simply one byte at a time, but that byte can have many purposes. It can be a simple number; it can be positive or negative (that is, be signed); it can represent a character, or it can be part of a memory address.

The memory addresses themselves are (from the processor's viewpoint) identical. However, their arrangement within the Color Computer is somewhat different and quite specific. Because of the synchronous address multiplexer (the SAM), the memory addresses (known as the memory map) are organized for special

* Describe relative addressing.

The mode in which the information is found relative to the position of the program counter.

* What are the levels of addressing?

Non-indirect and indirect.

* What does SAM mean?

Synchronous Address Multiplexer.

* What is found from $0000 to $7FFF in the Color Computer memory map?

RAM (read-write memory).

* What is found from $8000 to $9FFF in the map?

The Extended Color BASIC ROM (read-only memory).

* What is found from $A000 to $BFFF in the map?

The Color BASIC ROM.

* What is found from $C000 to $FEFF in the map?

Cartridge ROM, when plugged in.

* What is found from $FF00 to $FFFF in the map?

Vectors and SAM registers, control, ports, video graphics display, processor speed, video addresses, and other functions.

* What is assembly?

The process of converting source code (mnemonics) into object (binary) code.

* What is disassembly?

The process of translating binary code into a source (mnemonic) listing.

Course summary

* What does VDG mean, and what is its purpose on the Color Computer?

VDG means Video Display Generator; it is used for alphanumeric, semigraphic, and high-resolution graphic and color display.

* What does Hz mean? What does it mean when it is said that the Color Computer has a .89 MHz clock?

Hz means Hertz, clock pulses per second; a .89 MHz clock means a master set of pulses occurring approximately 890,000 times per second.

* What is a position independent program? What addressing mode is essential to position independent programming?

A machine language program designed to run correctly no matter where it is located in memory is position independent. Relative addressing is necessary for position independent programming.

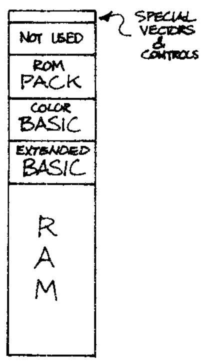
* What is an integer?

A number, positive or negative, which contains no fractional or decimal part.

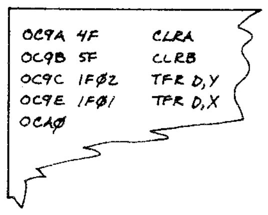
* What is a floating point number?

A number, positive or negative, which contains a fractional or decimal part.

purposes. From the start of memory to address **$7FFF** is reserved for read-write memory, or RAM; the next four blocks of memory (starting at **$8000**, **$A000**, **$C000** and **$E000**) are reserved for read-only memory, or ROM, and in the Color Computer are used for Extended BASIC, Color BASIC, and cartridge ROM. The last block is unused in the Color Computer. RAM may be substituted for ROM under certain conditions.



The last 256 bytes of memory are reserved for vectors and control, ports, video graphics display, processor speed, video addresses, and other functions. By writing information to the SAM, these functions can be turned on or off. Among the most important functions designed into the Color Computer are: control of the cassette and printer output; selection of 16 different low-, high-, and medium-resolution color graphics modes; RS-232 communications input and output; keyboard input; input from joysticks or other analog devices; control over the processor's clock speed; output of sound; determination of available memory and selection of the type of memory arrangement; control of and storage of vectors for interrupts.



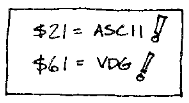
```
0C9A 4F     CLRA
0C9B 5F     CLRB
0C9C 1FØ2   TFR D,Y
0C9E 1FØ1   TFR D,X
0CAØ
```

Source code is normally assembled using an editor/assembler package, but hand assembly can be performed. For hand assembly, a list of opcodes and their respective hexadecimal equivalents is necessary. Also, it's essential to have a description of how each opcode works, the flags it affects, and how its operands are constructed and used.

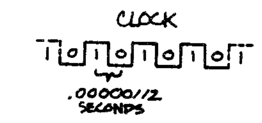
Assembly, whether by hand or using an assembler, is a two-pass process. During the first pass, the opcodes are assembled and put in place, and during the second pass the operands are created, calculated or otherwise determined directly from the operand information in the source listing, or from the labels used in the listing.

During the assembly process, the automatic assembler detects and reports errors. Hand assembly will reveal those opcodes or operands which are not permitted according to the information provided in the processor's data booklet.

Even correct source code can produce incorrect results, depending on the hardware configuration of the computer. In the case of the Color Computer, the most obvious conflict is with the standard ASCII codes and the video display generator, which uses a different arrangement of the four groups of 32 characters. These hardware conflicts are resolved by the programmer through debugging in combination with a careful reading of the software aspects of the hardware documentation.


$21 = ASCII !
$61 = VDG !

During hand assembly, the timing of instructions may be extremely critical. Especially during sound or communications processes, the timing of each instruction must be calculated. This timing is based on the computer's master clock frequency, which is specified as Hertz (Hz) or clock cycles per second. All the timing information is provided as part of the processor's data booklet. Some



timing is consistent with every occurrence of an instruction, other timing depends on the character of the operand.

The goal of position independent programming — that is, programs that will load and execute in any area of memory — can be achieved with the 6809 processor. Position independence is achieved using program-counter relative instructions (,PCR instructions), load-effective-address commands (LEA), long and short subroutine branches, and long and short program counter branches (simple, simple conditional, plus signed and unsigned conditional). By structuring the program around modular subroutines, both clarity and position independence can result.

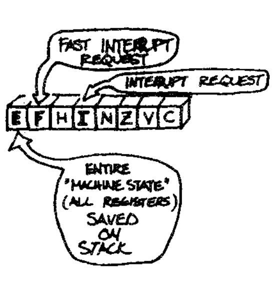
10110111.1-
01011010...

10110111.1-
01011010...

Among the less clear aspects of programming is the handling of floating-point numbers, that is those numbers consisting of both an integer and fractional part. The representation in the Color Computer is as a power of two exponent plus a four-byte mantissa. This achieves an overall accuracy of ten digits and an overall range of ten to the plus-or-minus 38th power.

EXEC ADDRESS
USR (ARGUMENT)

Using these numbers, and using BASIC at all, requires an understanding of its handling of free memory, how it loads machine-language programs, and the accessing of machine language programs via EXEC and USR. BASIC's USR command permits direct transfer of numerical or text information to a machine-language subroutine. BASIC's VARPTR command permits access to BASIC variables for use by a machine-language subroutine, and also provides a unique method of packing a machine-language program into a BASIC string variable in a program line.



The 6809 processor was created with interrupts in mind. Interrupts are hardware signals which cause the processor to set aside its current program and perform an interrupt service routine. Interrupts are use to provide accurate and program-independent timing and control functions. Hardware interrupts IRQ, FIRQ and NMI are used on the Color Computer; software interrupts SWI, SWI2 and SWI3 are used in ZBUG and in other kinds of program debugging, and for fast operating system subroutine calls on other kinds of computers.

Interrupts may be used for very fast timing, such as for synchronization with the video display. Video signals are used for interrupts on the Color Computer, and can be used as ordinary interrupts or in combination with the **SYNC** or **CWAI** commands for complete synchronization with the monitor picture.

The process of creating complete assembly language programs involves thinking the application through, creating a structure, writing modular subroutines, linking together the individual pieces, and debugging the whole.

Your Micro Language Lab course in Learning the 6809 is over, but your facility in programming has just begun. Now that you've reached this point, many earlier programs will

* What BASIC commands are used for accessing machine language programs? What does each mean?

EXEC, USR, DEFUSR, VARPTR, POKE and CLOADM. EXEC means execute a machine language program at the given entry point (starting address). USR means execute a machine language program, and transfer a variable from BASIC to it. DEFUSR defines a machine language program entry point (starting address). VARPTR means variable pointer, and is used to determine the position of a BASIC variable in memory. It can be used for packing machine language programs into BASIC string variables. POKE places a byte directly into memory. CLOADM loads binary information directly into memory.

* What are the 6809 interrupts?

Hardware interrupts NMI, FIRQ, IRQ and software interrupts SWI, SWI2, and SWI3.

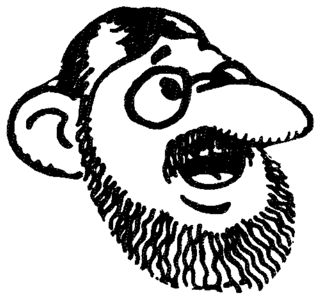
* What happens when an interrupt occurs?

The processor completes its current instruction, saves important machine information, and services the interrupt.

* What commands stop processor operation and wait for an interrupt?

SYNC and CWAI.

* Your course in learning the 6809 is now complete. I welcome your reaction, especially to this programmed learning section. Please send your comments to me, Dennis Kitsz, Green Mountain Micro, Roxbury, Vermont 05669.



begin to make more sense. Please review this course lesson by lesson, continue to use the question-and-answer programmed text in the margins, and try each of the example programs. The ability to program the 6809 — and all its smart cousins — is now yours.

I'm your programming guide, Dennis Kitsz. Good bye.