Inside the CoCo 3 — Hardware and Debugging

A house is on the screen. Your program just drew a beacon. Where did the dot go? Catch the machine between instructions, find the missing pixel in RAM, and show someone else how you know.



Illustration of the Tandy CM-8 monitor showing the green BASIC screen.

Brian K. Holman

Book 10 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Catch a write before it happens 3
Which RAM did I change? 6
Read the few instructions that mattered 8
Find the missing beacon 11
Put the beacon to the test 15
Recognize a device on the bus 19
Hand someone a reproducible bug 22
At the debugger bench 25
Publication and artwork notices 27

Catch a write before it happens

Chapter 1 of 7

A scratch byte starts at zero. Five instructions load its address, make a value, shift it, store it and return. Suppose you stop the processor on the store. Where would you look for the answer you just calculated? The register and the memory window will give you different answers, and both will be right.

Set up the workbench

Save any work in the guest. Use a fresh **Disk BASIC** session with **512 KB**, the **MC6809** and **RGB** video. These choices give us one reproducible memory map for the first five chapters. The two labs also work on the HD6309 in its startup mode.

Save pixel-mask.asm to your Mac. Choose **Develop → Assembly Development…**, then **File → Open Source…** and select your copy. Choose **Assemble**. In **Build details → Listing**, find the labels **shift**, **store**, **done** and **sample**. They name three instructions and one byte of data. The listing gives their addresses even if an edit moves them.

```
            org     $3f00
start       leax    sample,pcr  ; X points to our one-byte sample.
            lda     #$81        ; Two end bits are set: %10000001.
shift       lsra                ; Shift right. Which bit goes into carry?
store       sta     ,x          ; Set a breakpoint here, before the write.
done        rts                 ; Return to the caller with the byte stored.
sample      fcb     $00         ; Assembling and running reloads this zero.
            end     start
```

`LEAX` puts the address of **sample** in X. `LDA #$81` puts a value in A. In a logical shift right, bits move one place to the right, a zero enters at the left, and the old low bit moves into carry. Write the eight bits of $81 as **10000001**, then try the shift on paper. Keep a separate box for the bit that falls out.

Read the stopped instruction

1. In the assembly source, place the insertion point on line **8**, labeled **store**, and choose **Set Stop**. Choose **Run** and accept the load if prompted.
2. Open **Debug → Show Debugger**. In **Inspector**, find the instruction at the program counter, PC. For this listing it is **$3F07**, `STA ,X`. The comma and X mean “use the address in X.” The processor has reached the store and is waiting to execute it.
3. Read **A**, **X**, and **Carry** in the register panel. The shift produced A=$40 (01000000). Carry is set because the old low bit was one. X=$3F0A, the address labeled **sample**.
4. In **Inspector → Memory**, choose **Mapped CPU**. Type **3F0A** in the memory address field and choose **Go**. Addresses down the left mark the beginnings of rows, and the headers **00** through **0F** are offsets. At row **3F00**, column **0A**, the scratch byte is still **00**. The answer is in A. RAM still holds the byte assembled by `FCB $00`.

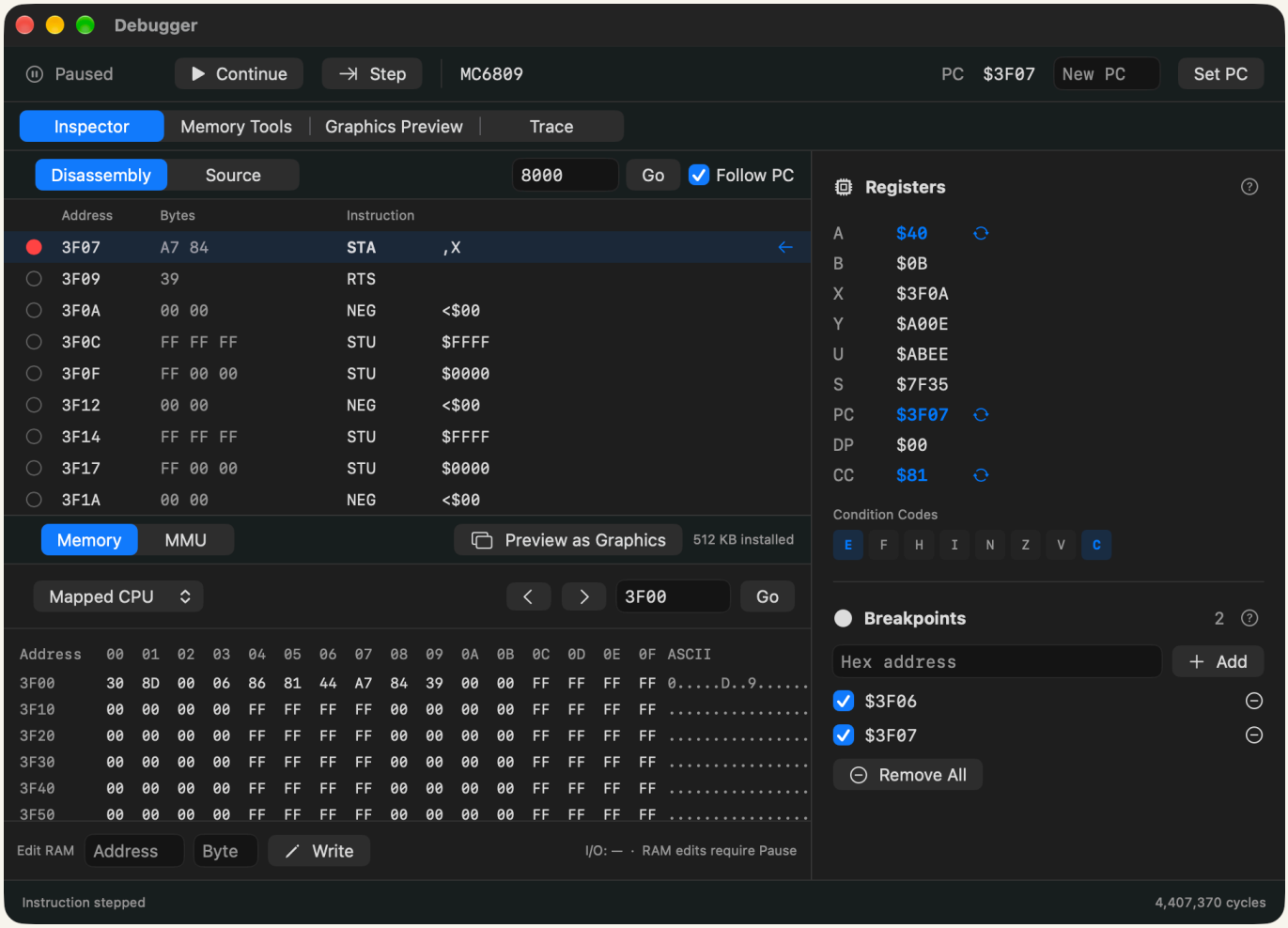


Figure 1: The paused Inspector points at STA at $3F07. The register pane shows A=$40 and X=$3F0A. The value in column 0A of row 3F00 is 00.

Give the processor one instruction

Choose **Step** once at the top of the machine debugger. Follow the three values that matter here.

Read here	Before Step	After Step
PC	$3F07, the store	$3F09, the return
A	$40	$40
Mapped byte at X ($3F0A)	$00	$40

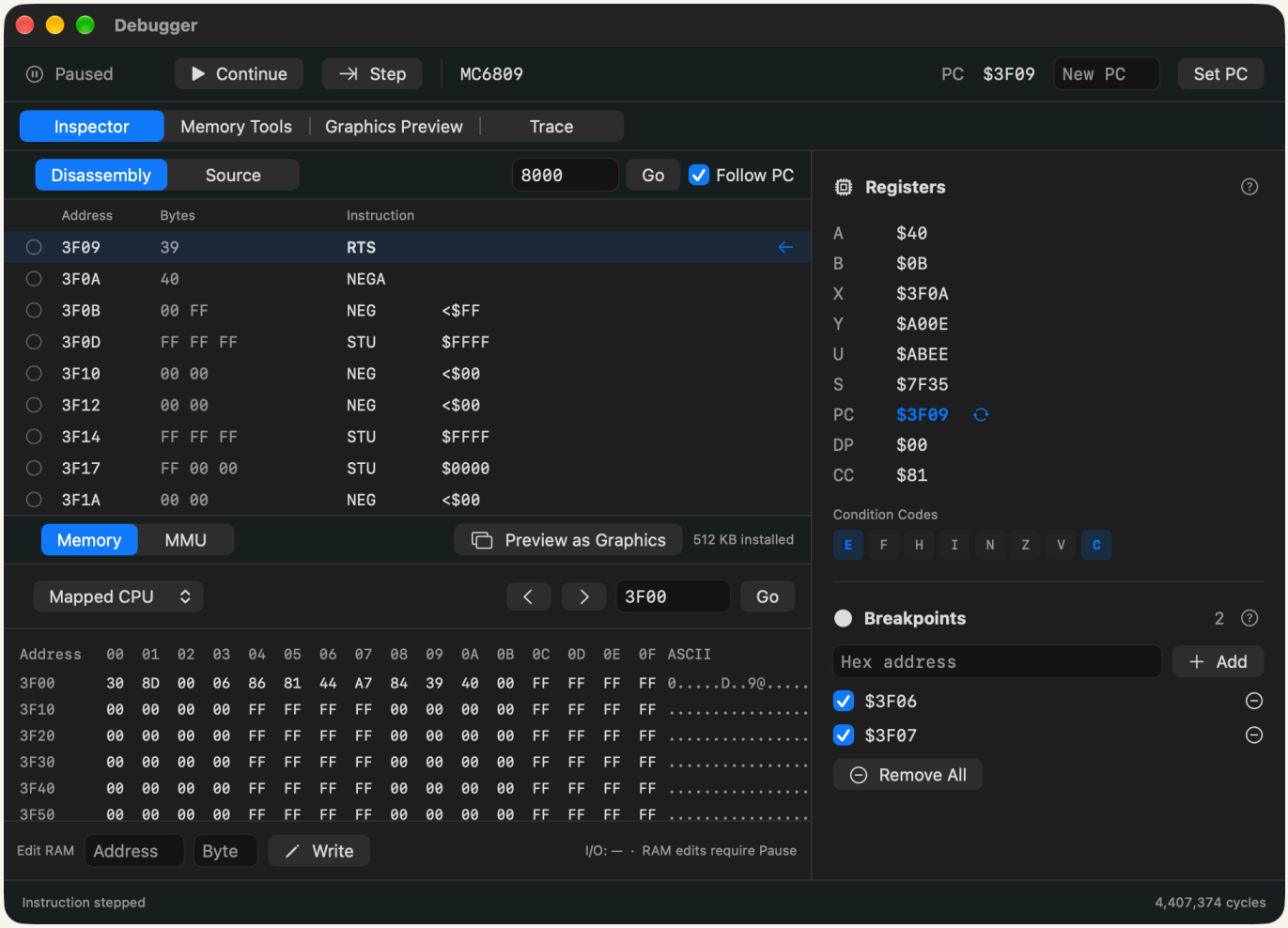


Figure 2: After one machine step the instruction arrow points to RTS at $3F09. Row 3F00, column 0A now contains 40. X still points to $3F0A.

A store copies A to the destination and leaves A available for another instruction. The PC advances by two because this encoding contains the store opcode and a second byte describing its indexed operand. If the byte in RAM already reads **40** when you first look, check the PC. At **done** you are one instruction later than **store**. Running the supplied binary again reloads the initial zero, so you can repeat the experiment.

Leave the guest paused on **done**. In the next chapter you can identify the location in the Mac emulator's model of the CoCo's RAM that actually holds this byte. For a refresher on the debugger's arrows and source stops, revisit Book 7's one-byte experiment.

Which RAM did I change?

Chapter 2 of 7

X points to $3F0A. That address fits in sixteen bits, just like every other address the 6809 can put on its address bus. Yet this machine has 512 KB of RAM, enough for eight complete sets of sixteen-bit addresses. Find the particular byte you just wrote. The memory management unit (MMU) inside the CoCo 3's GIME chip supplies the missing part of its address.

Translate one stopped address

Leave the processor at **done** from chapter 1. If you skipped ahead, run the scratch-byte exercise and Step once. Choose **Inspector → MMU**. Read the enabled/disabled indication, the selected task, and the row containing **$3F0A**.

The table has eight rows. Each covers $2000 bytes (8 KB) of CPU addresses. **Task 0** and **Task 1** are two sets of bank selections. In this Disk BASIC setup the MMU is enabled and task 0 is selected. The first row starts at $0000, so the row from **$2000 to $3FFF** is block 1. Its selected bank in this run is **$39**. Use the row for your own machine if it differs.

Question	Calculation in hexadecimal	Answer
How far is the byte from the start of its CPU block?	$3F0A − $2000	$1F0A
Where does physical bank $39 begin?	$39 × $2000	$72000
Which physical byte is that far into the bank?	$72000 + $1F0A	**$73F0A**

Look at the two results side by side. The CPU supplied the offset $1F0A within an 8 KB block. The MMU chose the physical block underneath it. This translation requires no instruction in the program. The processor uses X as usual when executing the store.

Confirm the address from both sides

1. Switch back to **Memory** and **Mapped CPU**. Enter **3F0A**, choose **Go**, and read column **0A** of row **3F00**. After the store it reads **40**.
2. Switch the address space to **Physical RAM**, enter **73F0A** and choose **Go**. Row **073F00**, column **0A** also reads **40**. The physical display may pad the address with a leading zero.
3. To test the connection, Continue to BASIC, keep the source stop on **store**, and Run the assembly again. The binary reloads **sample** as zero. At the stop read the same two addresses. They both read **00**. Step once and both read **40**. One guest store accounts for both changes.
4. Continue. The program executes its return. Click the CoCo display, type `PRINT 27` and press Return. BASIC should print **27** and its **OK** prompt.

Two unrelated RAM cells can happen to contain $40. What makes this a good diagnosis is that the active MMU row predicts the second address, and both views follow the transition from the assembled zero to the stored value. To see the *same* CPU address pick up an entirely different byte, follow Borrow block 2 in Book 9. That exercise deliberately changes one bank selection and restores it afterward.

Try this translation with a running OS-9 process in One address, two views.

Read the exceptions at the top of the map

Before using the same calculation elsewhere, read **Effective read** and **RAM backing** in the MMU table. They answer different questions. The scratch byte in this lab was in ordinary RAM. Other CPU addresses have other jobs.

What the inspector shows	What you can conclude
A RAM bank for the active mapping	The bank and offset locate the physical byte, as they did for **sample**.
A ROM overlay	CPU code can read ROM at that address while RAM exists underneath. The **RAM backing** button inspects the physical RAM. Its contents can differ from the mapped view.
The note for $FE00–$FEFF	The RAM-vector setting decides whether this range follows the final bank or uses fixed RAM. Read the current setting before translating here.
Device I/O in $FF00–$FFEF	An address can select hardware. The inspector displays a dash where it cannot safely supply a value. A dash does not mean the device returned zero.
System vectors in $FFF0–$FFFF	The CPU reads vector bytes here to find destinations. Read the inspector's system-ROM note when following one.

If the MMU is disabled, the inspector's **Effective read** column reports the fixed mapping. A number in an inactive task column describes what was programmed there; on its own it cannot tell you what a CPU read will see. This distinction also matters to breakpoints. The ordinary instruction breakpoint accepts a CPU address such as **3F07**. The much larger number **73F0A** identifies a byte in installed RAM.

For the circuitry behind the table, see Tandy Corporation, *Color Computer 3 Service Manual*, section V, figure 5-7, "MMU Block Diagram." Follow the CPU block selection on one side of that figure and the bank selection on the other.

Read the few instructions that mattered

Chapter 3 of 7

The two memory windows showed what changed during a store. How did $40 get into A in the first place? A trace keeps a record as the machine runs, so you can inspect a past instruction after BASIC has already returned. Give the recorder a start and an end. Otherwise this tiny program can disappear among thousands of instructions spent waiting for a key.

Arm a short capture

1. In **Assembly Development**, use the supplied pixel-mask.asm. Remove the source stop from **store**, on line 8. Set it on **shift**, on line **7**, and **Run** again. The only source stop for this pass should be **shift**.
2. In the machine debugger verify **Paused**, **PC $3F06**, **A $81**. If your addresses differ after an edit, use those for **shift** and **done** in your current listing.
3. Choose **Trace**. Enter **64** for **Sample limit**, **3F06** for **Start at**, and **3F09** for **Stop before**. Leave **Screen events** and **Scheduler events** off for this short run. **Start New** replaces the previous capture, so export an earlier trace first if you want to keep it.
4. Choose **Start New**, then **Continue**. The program executes its shift, stores the byte and returns to BASIC. The trace status should say **Stop address reached (before instruction)**. The emulator can now show **Running** and a PC somewhere in ROM. Click the CoCo display, type `PRINT 27`, and press Return. New BASIC input works while the recorded samples remain available.

Read an instruction from beginning to end

Select the **LSRA** sample at **$3F06**. Look in the detail area below the table. The two register lines belong to the selected instruction. An initial sample can also show $3F06 when resuming from a breakpoint. Pick the row where the before and after PC differ and the cycle count is two.

Selected instruction	Before	After	Instruction cycles in this lab
LSRA	PC $3F06, A $81	PC $3F07, A $40, carry set	2
STA ,X	PC $3F07, A $40, X $3F0A	PC $3F09, A $40, X $3F0A	4

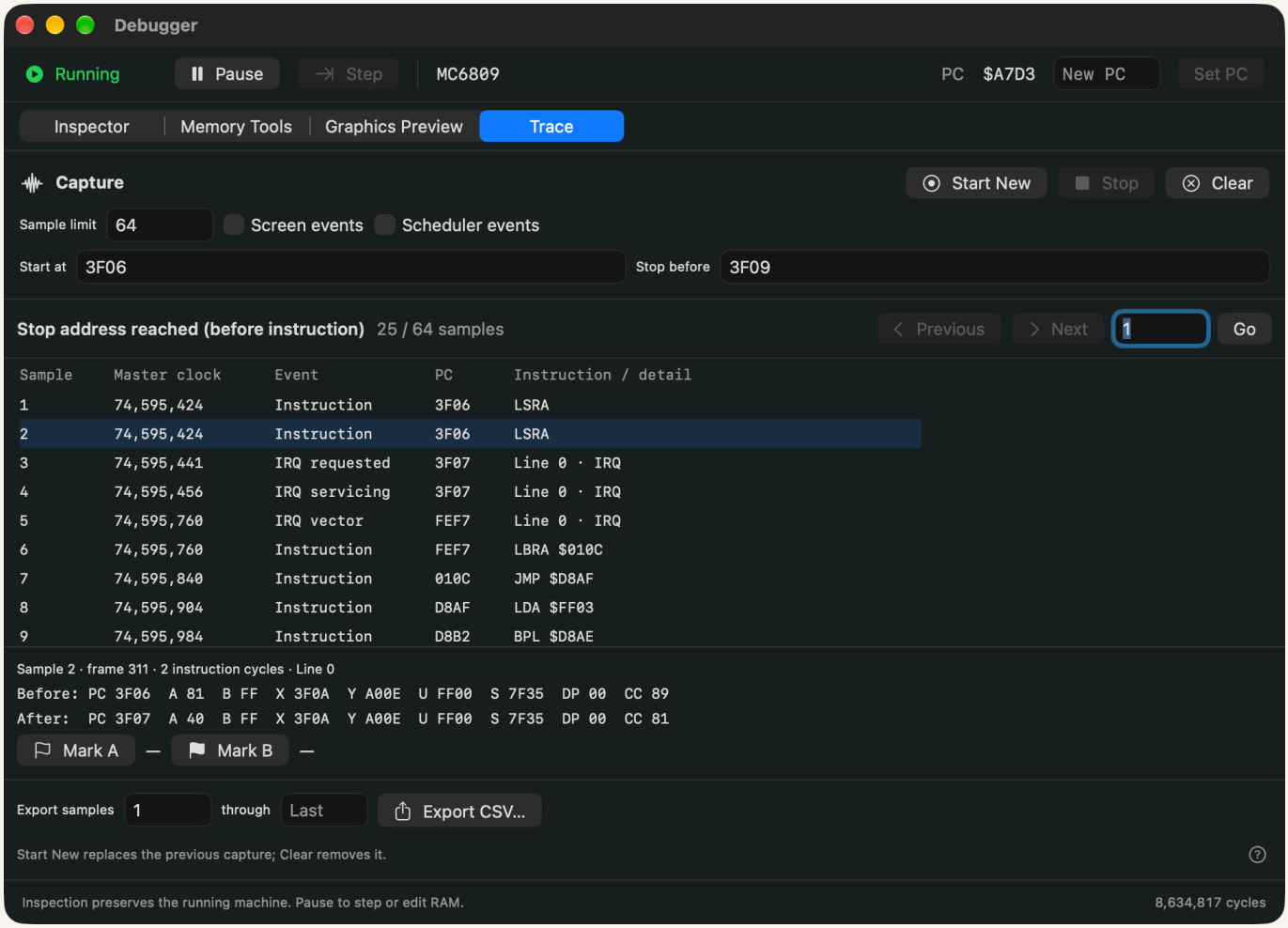


Figure 3: The trace is complete and the emulator is running. The selected LSRA sample shows two instruction cycles, with PC changing from 3F06 to 3F07 and A from 81 to 40.

Double-check the instruction name as well as the address when locating **STA ,X**. The table can contain several kinds of events at one PC. The unchanged A and X on the store row are exactly what you saw in chapter 1. The trace gives you the instruction, its operands and its register states. The before-and-after memory check supplies a separate observation of the destination byte.

Account for the extra rows

In the pictured capture the CPU completed the shift, then received an **IRQ** before it executed the store. You can see **IRQ requested**, **IRQ servicing** and **IRQ vector**, followed by instructions at other addresses. The interrupt handler eventually gave the interrupted code its turn again. Find **STA ,X** by address and mnemonic lower in the list. Another run can have fewer rows, or service the interrupt at a different boundary. The order of the program's shift and store stays the same.

The **Master clock** column places events on the emulated machine's timeline. The detail beneath the selected instruction gives that instruction's CPU-cycle count. Use the latter to compare the **2** and **4** above. The elapsed time on your Mac also includes the emulator, its windows and your time reading them. It cannot measure the cost of a CoCo instruction.

A sample limit reached before the desired stop, a previous capture still displayed, and a stray breakpoint can all produce a plausible-looking short list. Check the capture status, the start and stop addresses, and that you chose **Start New** *before* Continue. If needed, re-arm from the source stop and allow more samples. Once you have the

two instruction rows, remove the source stop from **shift**. You can now use the machine for the BASIC graphics lab.

Find the missing beacon

Chapter 4 of 7

The CoCo draws a house. Six lines later BASIC draws a dot inside a rectangular receiver, but you can still see the house. Did BASIC miss the dot, or are you looking at the wrong picture? Catch the program just before it draws the dot. You can answer both questions without letting it switch the monitor.

Stop on either side of PSET

Use a fresh **Disk BASIC** session with **512 KB** of RAM and **Settings → Video Signal → RGB Monitor**. Open BEACON.bas from the Manual Library. In a browser, save the file and open it in **Develop → BASIC Development**. The program and its graphics pages will live in the practice machine's RAM.

```
10 WIDTH 32:PCLEAR 4:CLS
20 PMODE 1,1:PCLS 0:SCREEN 1,0
30 COLOR 1,0:LINE(32,80)-(96,112),PSET,B
40 LINE(32,80)-(64,48),PSET:LINE(64,48)-(96,80),PSET
50 PMODE 1,3:PCLS 0:COLOR 3,0
60 LINE(144,48)-(184,80),PSET,B
70 PSET(162,64,3)
```

Line 10 reserves four BASIC graphics pages. One PMODE 1 picture takes two of those pages. Line 20 picks page 1 as the start of the first picture and uses `SCREEN` to display it. Lines 30 and 40 draw the walls and roof. At line 50, BASIC starts drawing at page 3. Notice which command from line 20 is missing on line 50. The monitor carries on showing the house while BASIC clears the other picture and draws its receiver.

1. In BASIC Development, click the breakpoint gutter beside **70** and **80**, then **Run**. Accept the program load if asked. When line 70 is highlighted, the receiver outline already exists. The `PSET` on the highlighted line still lies ahead.
2. Open **Debug → Show Debugger → Graphics Preview**. **Current Video Page** shows the house. The information beside the picture reports **Physical RAM $070E00** in this setup.
3. Choose **At Address → Mapped CPU**, enter **1A00**, and choose **Go**. Select **PMODE 1** and **Set 0**. Now you should see a rectangular outline. The address gives the debugger a starting byte. The mode tells it how to turn the following bytes into pixels.
4. In **Pixel Inspector**, enter **81** for X and **32** for Y, then press Return. Write down the **Address**, **Byte**, and **Bit mask**. In BASIC Development choose **Continue**. At line **80**, go back to the same preview and read those three fields again.

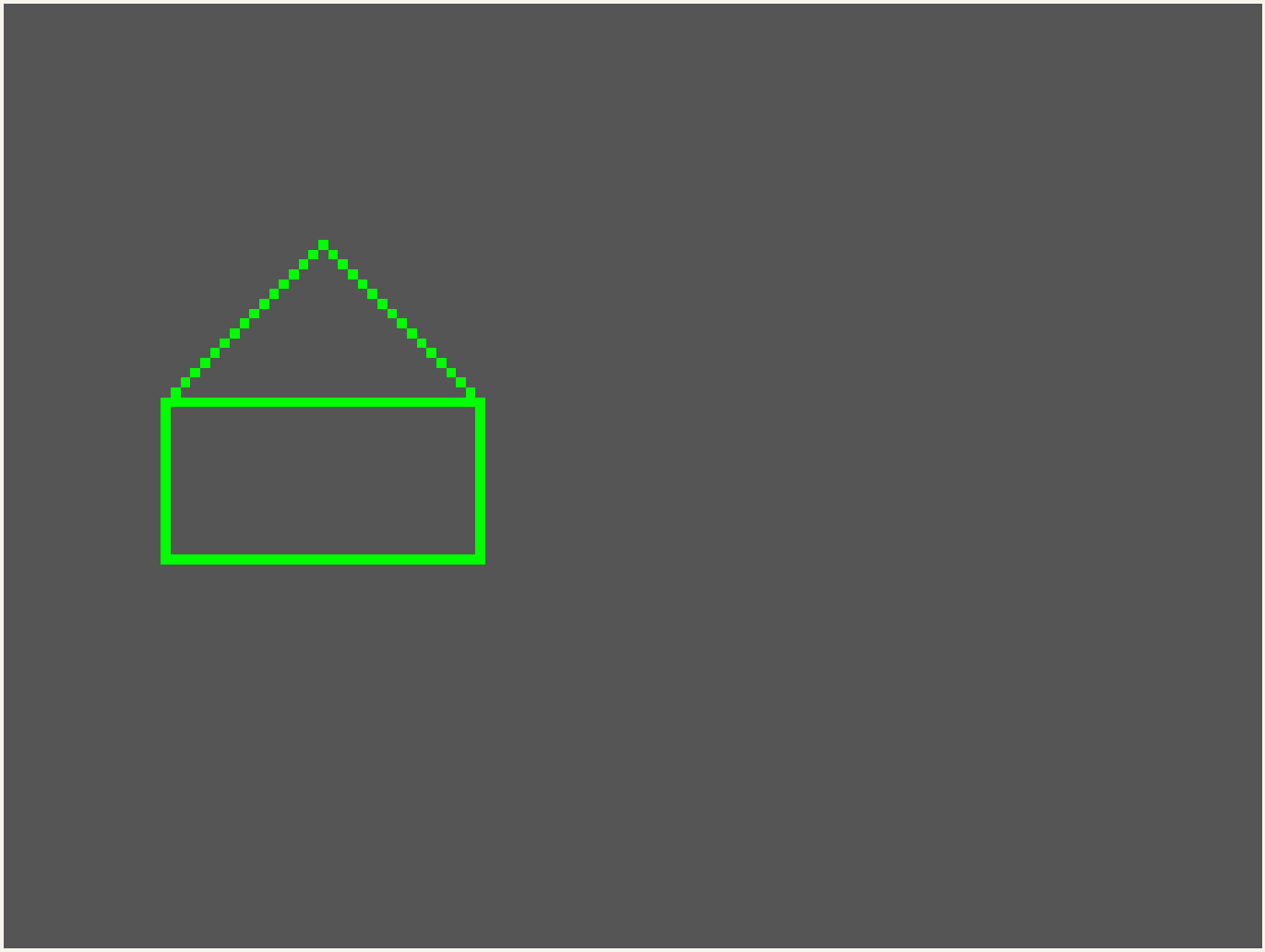


Figure 4: Current Video Page at line 80. A green house stands at the left.

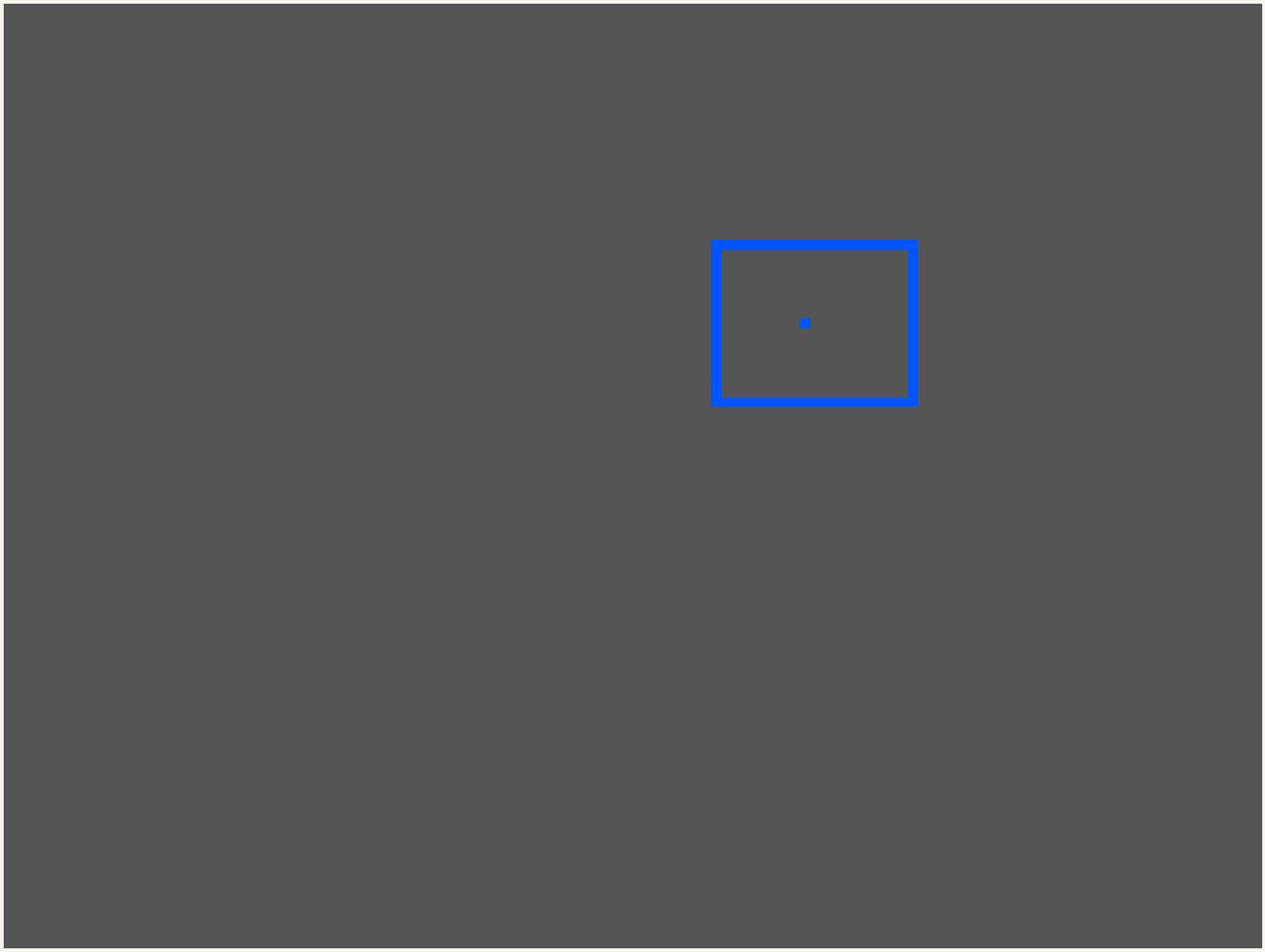


Figure 5: At Address at the same stop. A blue rectangular receiver at the right contains one blue dot.

These are the two 128 × 96 preview buffers from a single pause. In the app, the pixel-selection outline can cover part of the dot. Use the **Byte** field to tell whether it changed.

Work back from the pixel to its byte

BASIC used coordinates **(162,64)**. PMODE 1 represents its picture with **128 columns and 96 rows**, half as many positions in each direction as BASIC's drawing grid. On this grid the dot lands at **(81,32)**. The inspector's X and Y fields take these stored pixel coordinates in decimal.

Starting at the top left of the hidden picture	Distance in bytes
One row contains 128 pixels. Four two-bit pixels fit in each byte.	128 ÷ 4 = 32 bytes per row
Skip the first 32 rows to reach Y = 32.	32 × 32 = 1,024 bytes, or $0400
Skip 20 whole groups of four pixels to reach the group containing X = 81.	20 bytes, or $14
Add the two offsets to the mapped start address.	$1A00 + $0400 + $14 = **$1E14**

Read the pixels from the high pair of bits toward the low pair. Byte 20 holds X positions 80, 81, 82 and 83. The dot at 81 uses the second pair, bits 5 and 4. The inspector's mask **$30**, binary **00110000**, picks out that pair.

Stop	Mapped byte at $1E14	Four pixel codes, left to right
Before line 70	$FF	11 **11** 11 11
Before line 80	$EF	11 **10** 11 11

The pen number **3** in the BASIC source produced the stored two-bit code **10** in this run. Pen numbers, packed pixel codes and RGB colors describe different stages of drawing. You just measured the stored code directly. The other three pairs survived the write.

Find that byte in physical RAM

With the mapped preview selected, choose **Show Byte in Memory**. The Inspector opens to mapped RAM. Read row **1E10**, column **04**. Open **MMU** and look up the CPU window $0000–$1FFF. On this Disk BASIC machine, task 0 is active and the window uses bank $38. Its physical base is $38 × $2000 = $70000. The hidden picture therefore starts at **$71A00**, and its marked byte is **$71E14**.

Go back to **Graphics Preview** → **At Address**, choose **Physical RAM**, enter **71A00**, and choose **Go**. Leave **PMODE 1**, **Set 0** and the pixel coordinates as before. The receiver is still there. The inspector now names the marked byte by its physical address. **Current Video Page** still reads from **$70E00**. The CPU drew the dot, and the video hardware was reading a different region of RAM.

If you see the house in both previews, check the address *space* as well as the digits. If you see an unmarked rectangle at line 70, you have reached the intended first stop. Use **Continue** to reach 80 before looking for $EF. Leave the machine paused there for the one-byte experiment.

Put the beacon to the test

Chapter 5 of 7

You have an explanation for the missing dot. Now use it to predict an outcome the screen can confirm. If $1E14 really supplies the beacon on the hidden page, you should be able to erase the dot, keep the surrounding rectangle, and leave the house on the monitor throughout.

Edit one byte and put it back

Start at BEACON's line-80 stop. Leave BASIC paused. If you are coming back to the book after a break, run the lab again so the same program owns this RAM. Before writing anything, check for the house on the monitor and the expected $EF byte on the hidden page.

1. Choose **Graphics Preview → At Address → Mapped CPU**, enter **1A00**, choose **Go**, and select **PMODE 1**, **Set 0**. Select pixel **81,32** and press Return. Choose **Show Byte in Memory**. Check the highlighted byte at **$1E14**.
2. In **Edit RAM** at the bottom of the mapped Memory pane, enter **1E14** for the address and **FF** for the byte. Choose **Write**. In **Graphics Preview**, inspect the same hidden page. All four pixel codes in the byte now read **11**. The dot takes on the surrounding background. The rectangle survives, and the house still fills the CoCo display.
3. Choose **Show Byte in Memory** again. Enter **1E14** and **EF** in **Edit RAM**, then **Write**. Return to the hidden preview. The dot is back inside the rectangle. Leave **EF** in place so the program can show you the completed picture.

The edit controls live in **Mapped CPU** memory and require a paused machine. **Physical RAM** provides a second, read-only view. If you jump from a physical graphics preview into Memory, go back to the mapped preview before following step 1. Also check the address in **Edit RAM** itself. The **Go** field above the grid controls which bytes you can see.

Let the monitor read the other page

```
80 A$=INKEY$:IF A$="" THEN 80
90 SCREEN 1,0
100 A$=INKEY$:IF A$="" THEN 100
110 PMODE 1,1:SCREEN 1,0
120 A$=INKEY$:IF A$="" THEN 120
130 SCREEN 0,0:WIDTH 40:CLS
140 PRINT "BEACON CHECK COMPLETE"
150 END
```

While BASIC is still stopped at 80, click the gutter dots to remove both breakpoints. Choose **Continue**, then click the CoCo display to give the guest keyboard focus. Tap and release Space once per picture, waiting to see the change before tapping again.

After this tap	BASIC executes	Look for
First	Line 90 selects the current drawing page for display.	The receiver and its dot replace the house.
Second	Line 110 selects page 1 and shows it.	The complete house returns. There is no second set of roof-drawing instructions.
Third	Line 130 restores text and selects 40 columns.	**BEACON CHECK COMPLETE** followed by **OK**. Enter `PRINT 2` and press Return. BASIC answers **2**.

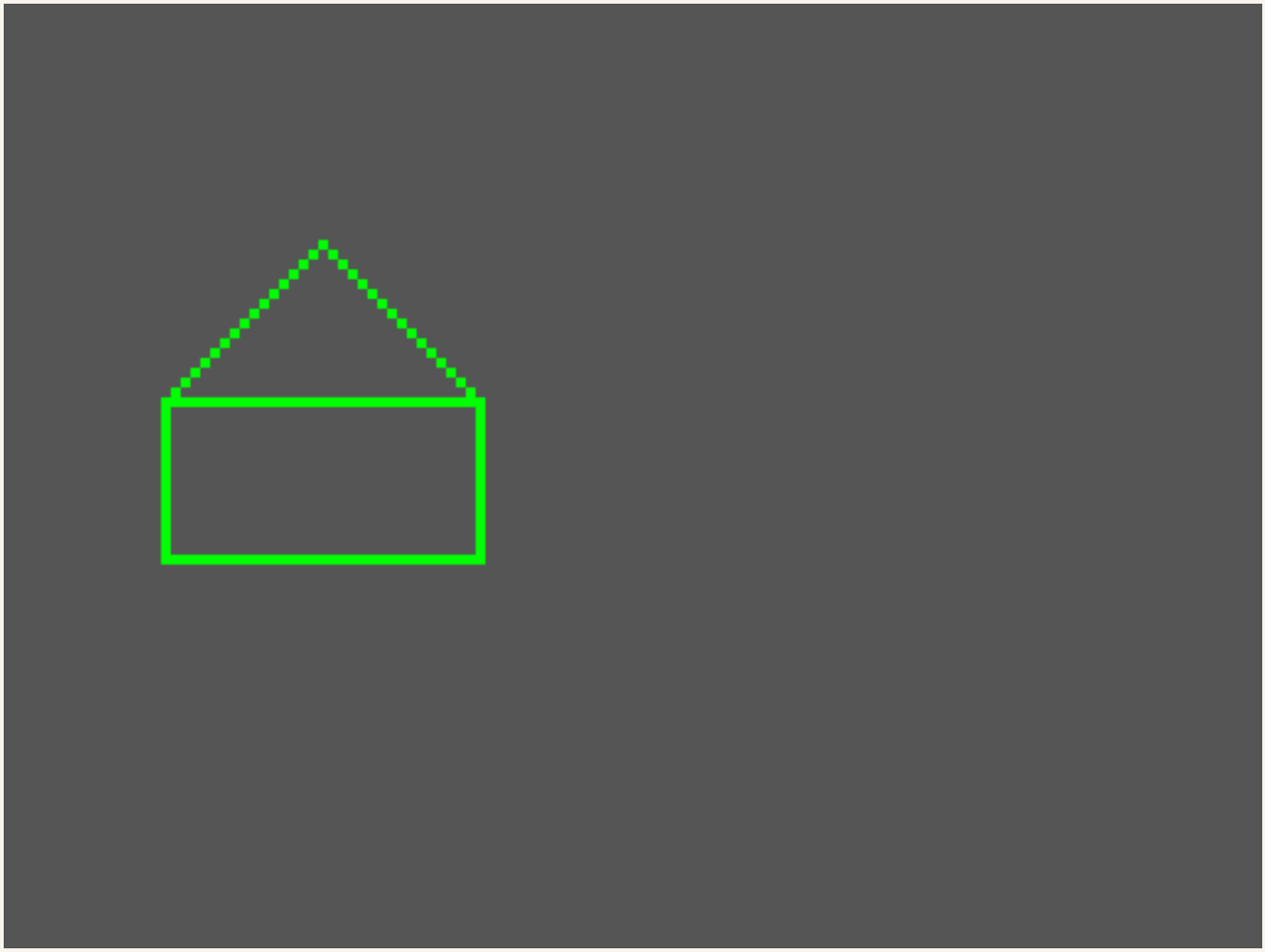


Figure 6: CoCo monitor before the first tap. A green house appears at the left on a gray background.

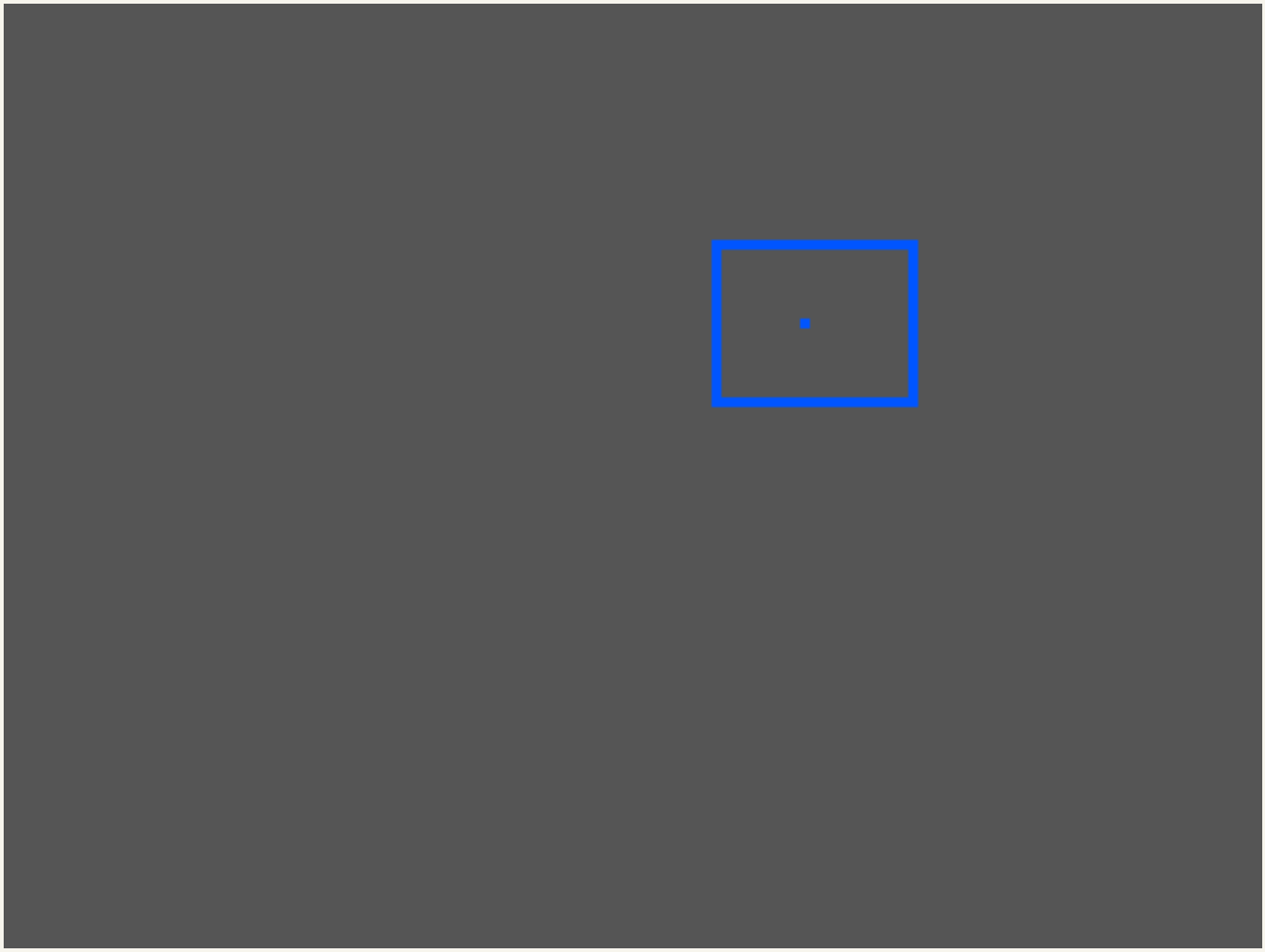


Figure 7: CoCo monitor after the first tap. A blue receiver at the right encloses the restored dot.

Each picture above is an actual capture of the emulator's active display. The receiver matches the debugger's earlier view of the hidden page. The program drew no new pixel between the first tap and this reveal. If the first tap appears to do nothing, check that BASIC is running and give the CoCo display focus again. Its `INKEY$` loop waits there as long as it needs to.

Tell three kinds of picture change apart

Suppose someone reports, "The color on my screen changed." Which observation would tell you *how* it changed?

What changed	What you could measure
A pixel was drawn.	At lines 70 and 80, the hidden page's base stayed the same. One byte changed from $FF to $EF.
A different picture was displayed.	After the first tap, the current physical video base moved from $70E00 to $71A00. The dot was already in the second picture.
A palette slot was given a different color.	In the CoCo 3 assembly starter, the write to $FFB6 changes the color in GIME palette slot 6. Pixels or text cells that already select that slot can change appearance together.

For the third case, follow the Earth or Mars palette write. You can stop at the store and compare the register with the selected palette slot, just as you compared a register with a RAM byte at the start of this book. For a CoCo 3 graphics example that sets its palette from BASIC, see Star Chart.

Recognize a device on the bus

Chapter 6 of 7

A store instruction gives the machine an address and a value. At $3F0A, a RAM chip kept the value. What listens at $FFB6? Run a program whose choice you can see on the monitor, then find the instruction that made the choice visible.

Catch a write to the GIME

Start at the prompt in a fresh 512 KB Disk BASIC session with RGB video. In **Develop → Assembly Development...**, open **File → Hello World — Earth or Mars** or save the assembly listing. Set a breakpoint on **greet**, at `STB BG6`. Run, click the CoCo screen, and press **2** for Mars. The program has chosen a sentence but has yet to print it.

1. In **Debug → Show Debugger**, choose **Inspector**. PC is $3F22 in the supplied build and B contains $25. Find the definition of **BG6** in the source. It names $FFB6, GIME palette slot 6 (the seventh slot, counting from zero).
2. Inspect **Mapped CPU** memory at **FFB6**. The low six bits read $09 in the fresh practice setup. Step one machine instruction. PC advances to $3F25 and the low six bits now read $25, the value B supplied.
3. Continue and read **Hello Cold Red Dot World!** on a red background. How did the letters acquire that color? The next instructions set a text attribute that selects slot 6 as the background. The output routine writes the characters. You can follow that sequence in Earth or Mars.

The CoCo 3's GIME chip controls both the palette and the mapping from CPU addresses to RAM. You met its memory mapping earlier. This time the CPU used an address on its bus to select a color register. An address is how the CPU reaches something. Knowing the type of hardware at that address tells you what the write means.

Read the switch and the running connections separately

In the practice machine, open **Cartridge → Cartridges...**. Confirm that the FD-502 occupies Multi-Pak slot 4, the other three slots are empty, and both **Live routing** indicators show slot 4. Leave the disk idle. This experiment keeps the ROM accessible while briefly choosing an empty slot for floppy I/O.

At the BASIC prompt enter `POKE &HFF7F,&H32`. Watch the live **ROM** selection stay on 4 while **I/O ports** changes to 3. Before doing anything with the disk, enter `POKE &HFF7F,&H33` to restore both live connections to 4. The startup selector stays at 4 throughout. See the two selections in the cartridge window.

Why $32? The two selector fields count from zero. In $32, the high field contains 3 (the fourth slot for ROM) and the low field contains 2 (the third slot for I/O). Changing the low field back to 3 reconnects the controller's ports. We learned the result by watching the routes. In the debugger, mapped $FF7F shows a dash because the inspector does not perform a guest read there. A dash means unavailable. The guest can still write the register, as you just proved.

Follow the interruption in a trace

If the pixel-mask recording is still in Trace, select it. Otherwise, repeat the short capture or use the figure below. A hardware IRQ can arrive at a different point on another run. Read the **Event** column before interpreting a PC as an instruction that ran.

In this capture	What happened
IRQ requested, PC $3F07	The request arrived while the next application instruction was the store.

In this capture	What happened
IRQ servicing, PC $3F07	The CPU accepted it. An IRQ request can instead be masked, so the first row alone cannot establish that a handler ran.
IRQ vector, PC $FEF7	The CPU obtained the handler entry. Inspect the mapped bytes at $FFF8 and $FFF9. The pair FE F7 is the address $FEF7, high byte first.
Instruction, PC $FEF7	The handler's first instruction branches to $010C. The following jump leads into Disk BASIC's IRQ code.

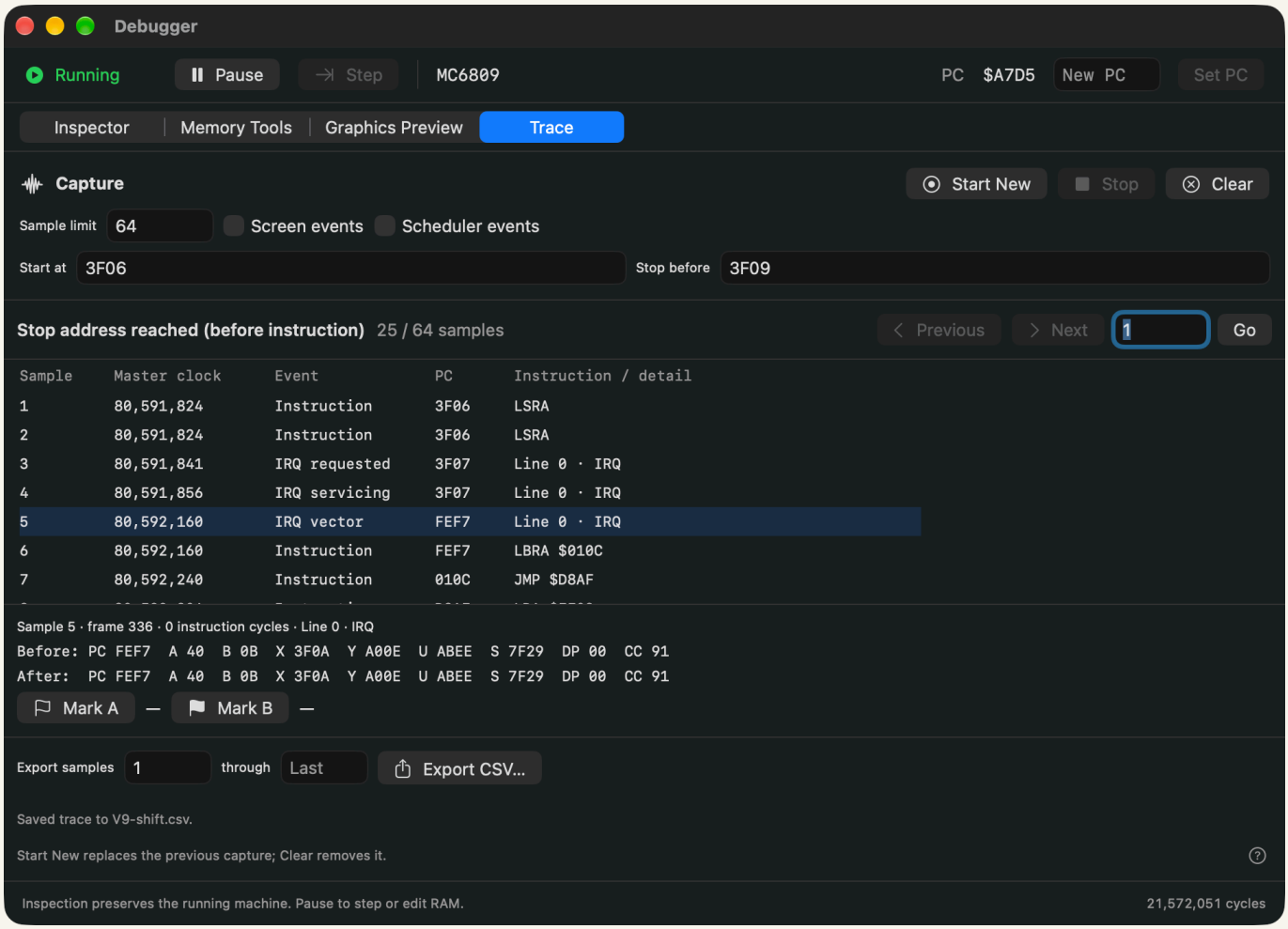


Figure 8: Native Trace shows IRQ requested, IRQ servicing and IRQ vector before the handler's first instruction at FEF7.

The selected row describes the vector event. Its zero in the instruction-cycles field does not time the entire interrupt. Look farther down the capture. The instruction **RTI** restores the saved context and its next PC is $3F07. The four-cycle **STA ,X** finally runs there with A=$40. Compare with the deliberately triggered software interrupt, where the program controls the instant of entry.

Some device reads have effects of their own. Disk BASIC's handler in this trace reads a PIA, the chip handling peripheral I/O. The debugger shows a dash at mapped $FF02 to avoid doing an extra read merely because its memory window is open. Use captured instructions, registers and device indicators to learn what the guest did.

Read the register set for this CPU

Both guest CPUs reached the same palette store and responded to the same Multi-Pak writes. A, B, X, Y, U, S, PC, DP and CC give you a common set of landmarks. On an HD6309, the Inspector also offers E, F, V and MD. E and

F make up W, just as A and B make up D. Read MD when comparing operating modes. The HD6309 practice runs in this book use its startup emulation mode. Book 9 lets you watch W in a transfer and inspect a native-mode change.

Hand someone a reproducible bug

Chapter 7 of 7

"Sometimes the picture is wrong" gives the next investigator a lot of work. "My program reached line 80, and the byte for the new dot changed on the other page" gives them an experiment. Make a small evidence packet that a second reader could reproduce from a fresh machine.

Reproduce the observation

Start a fresh **Disk BASIC, 512 KB, RGB** session. Record which guest CPU you selected. Open BEACON.bas in BASIC Development, set breaks at 70 and 80, and Run. The monitor shows the house. The stop at 70 is **before** PSET. Use **Continue** once to reach 80, where the draw has happened. Save both observations, including the unchanged monitor and the hidden page viewed at mapped $1A00.

To capture the bytes around the beacon, take these steps **at each stop**:

1. Open **Debug → Show Debugger → Memory Tools**. Select **Mapped CPU**. Enter start **1E10** and end **1E1F**. Both endpoints count, so the file will contain sixteen bytes.
2. Choose **Export Binary...** and give each stop a distinct name, for example **beacon-before.raw** and **beacon-after.raw**. Pause at the stated line while saving. An export captures memory when requested. An older disassembly left on screen can describe an earlier instant.
3. At line 80, repeat the export with **Physical RAM**, start **71E10**, end **71E1F**. In this setup it agrees byte for byte with the mapped after-file. If another setup maps block zero differently, compute the address from its MMU table first.

Stop	Sixteen captured bytes, first to last
70, before PSET	FF FF BF FF **FF** FF FF BF FF FF FF FF FF FF FF FF
80, after PSET	FF FF BF FF **EF** FF FF BF FF FF FF FF FF FF FF FF

Count the first byte as offset zero. Offset 4 means $1E10 + 4 = $1E14, the same answer the pixel calculation gave. The surrounding BF bytes belong to the receiver drawing and appear in both files. Record the result in your report: **one changed byte, one changed hidden pixel, same displayed house**. Clear the breakpoints and Continue. Three separate Space taps reveal the receiver, bring back the house and return to BASIC. At OK, enter `PRINT 2`.

Give physical code a CPU address

Repeat the assembly warm-up with pixel-mask.asm, stopping on **shift**. How could another reader tell which instructions lay next to this stop? The debugger can show the actual bytes in the backing RAM alongside the addresses the CPU used.

Choose **Memory Tools → Physical RAM**. In the practice configuration enter **73F00** through **73F09**, and give **Assumed CPU origin** the value **3F00** from the listing. Leave OS-9 interpretation off and choose **Disassemble**. Read the first row across. The physical address $073F00 holds bytes **30 8D 00 06**. The CPU saw those bytes at $3F00 as **LEAX 6,PCR**. Its PC after fetching the four-byte instruction is $3F04, so the effective address is $3F04 + 6 = $3F0A. The five decoded instructions end with RTS at CPU $3F09.

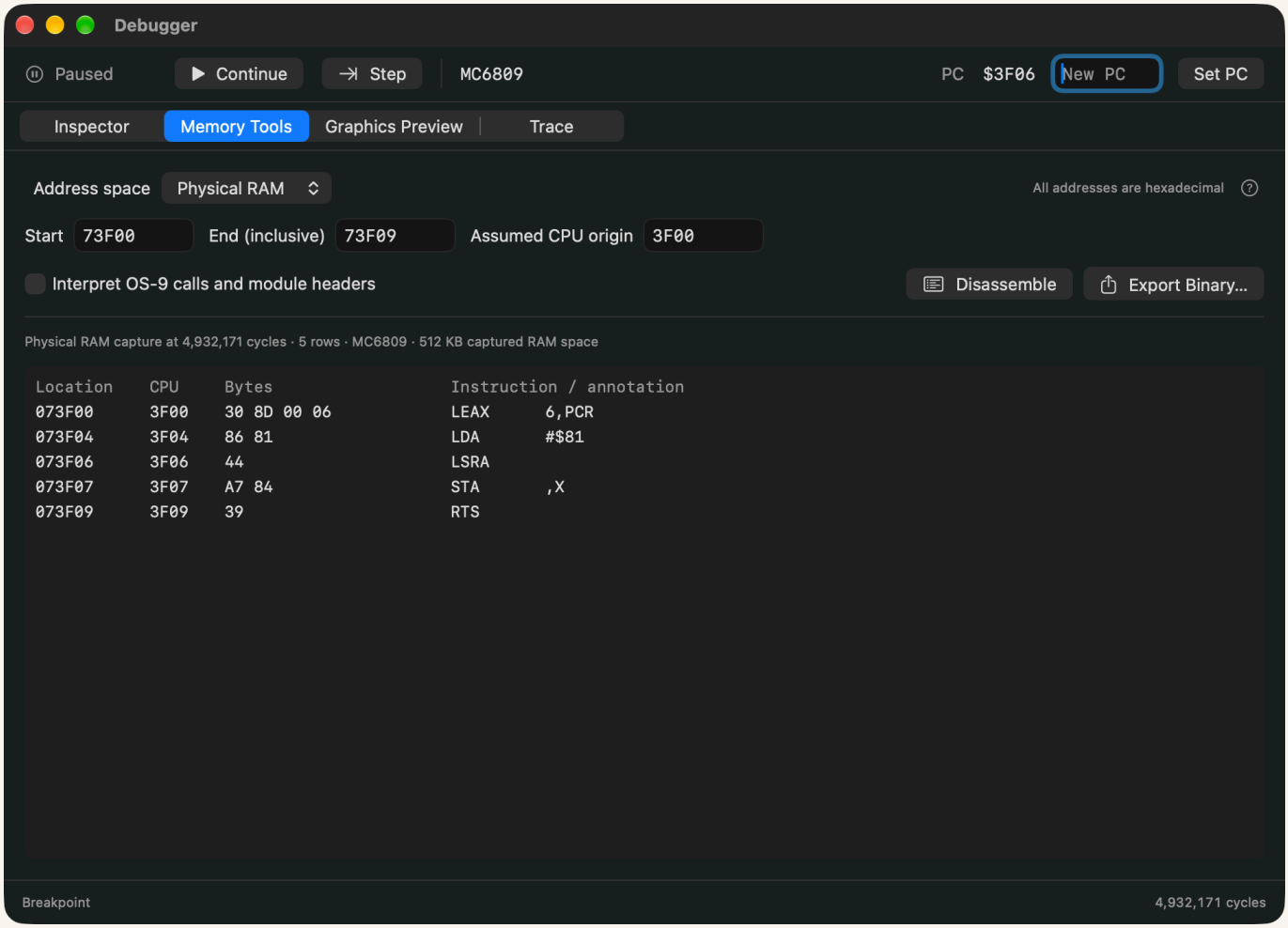


Figure 9: Memory Tools decodes five instructions from physical 073F00 through 073F09, with CPU origin 3F00. Export Binary is available above the rows.

The origin supplies the CPU coordinates for this interpretation, including relative branch targets. Changing it does not relocate the program or alter the file. For a direct check, choose **Export Binary...**. The ten bytes saved by the native app were **30 8D 00 06 86 81 44 A7 84 39**. The next byte, the data at $3F0A, falls outside your range. A raw capture carries no start address or Disk BASIC LOADM record. Include the address space and range in your notes. To give someone a runnable program, use the Assembly workspace's binary export instead.

Give the trace a readable key

At the shift breakpoint, open **Trace**. Use the earlier settings: 64 samples, start at **3F06**, stop before **3F09**. Choose **Start New**, then **Continue**. Save any trace you want to keep before starting another one. When the status reports the stop address, choose the inclusive first and last sample numbers under **Export samples** and click **Export CSV...**. Leaving the last field blank selects through the current last sample. For the illustrated run, 1 through the last sample preserved the interruption and the return.

A spreadsheet's **pc** column gives 16134 for the shift because address and register numbers in the CSV are decimal. The native window formats the same PC as hexadecimal $3F06. **bytes** is a hexadecimal instruction-byte string. Here are the two executed instruction rows to find in your own export:

instruction	pc → next_pc (decimal)	before_A → A (decimal)	bytes	cpu_cycles
LSRA	16134 → 16135	129 → 64	44	2

instruction	pc → next_pc (decimal)	before_A → A (decimal)	bytes	cpu_cycles
STA	16135 → 16137	64 → 64	A784	4

Select the LSRA row where PC advances. A zero-cycle boundary row can have the same mnemonic when execution resumes from a breakpoint. An IRQ can change the sample numbers and the distance between the rows. Use the PC, instruction bytes and register transition to identify them. The guest can already be back at BASIC while you inspect this file. Enter `PRINT 27` to check.

Write a report someone can repeat

Give the other reader the source you actually ran, the VCC Plus build, guest processor, profile, RAM, video setting and any relevant media assignments. Name the breakpoint and whether it is before or after the write. For raw captures include the address space and inclusive range; for physical disassembly include the assumed CPU origin. Give one expected result, one observation and a way to distinguish the explanations.

For the beacon, a useful closing sentence is: **“At line 80 the receiver contains the dot at (81,32), but the monitor still shows the house. One Space tap selects the receiver and the dot appears.”** That last action lets a reader distinguish a display-page choice from a failed draw without changing the program.

At the debugger bench

Appendix A

Leave this page open next to a paused machine. Start with the question you have about the program, then pick the view that can answer it.

Which view answers my question?

Your question	Where to look	What to read there
What will execute if I step once?	**Debug** → **Show Debugger** → **Inspector**	Read PC and the instruction at the arrow. At a store breakpoint, the register can already hold the new value while RAM still holds the old one.
What already executed while I let the machine run?	**Trace**	Select a recorded instruction and compare its before and after states. The shift trace shows both the input and the result.
Where did a CPU write land?	**Inspector** → **MMU**, then **Memory**	Read the active task, CPU block and bank. Add the byte's offset to the physical bank base. Check the effective mapping and compare both views.
Where is the picture that BASIC drew?	**Graphics Preview**	Compare **Current Video Page** with **At Address**. Choose the matching graphics mode and color set. Use Pixel Inspector to locate an individual field within a byte.
Why did the picture change when the pixel byte stayed the same?	**Graphics Preview** and the program's screen or palette instructions	The program can display a different page or change the color assigned to a pixel code. Flip the page, then compare the palette register example.
What should I send someone who wants to reproduce this?	**Memory Tools** and **Trace**	A raw range preserves the bytes. A trace records what ran. Write down the address space, range, stop and setup so the files have meaning outside your session.

Put the two exercises on your bench

pixel-mask.asm is the short assembly experiment. In the native reader, save a working copy, then open it in **Develop** → **Assembly Development...**. Assemble the supplied source as written for the first run so the named instructions and the addresses in the book agree.

BEACON.bas opens an editable BASIC example from the native reader. If you are reading in a web browser, save the file and open it in **Develop** → **BASIC Development...**. Opening the example lets you inspect its source. Choose **Run** when you are ready to replace the practice machine's BASIC program.

Both exercises use **Disk BASIC**, **512 KB**, **RGB**. Save your current program before switching setups. The two CPUs give the same checkpoints in these exercises. A recording can contain different interrupt rows from one run to the next. Compare the named instructions and their results first.

When your stop looks different

The byte is already $40. Look at PC. If it points to RTS, the store has executed. Re-run the assembly source with the breakpoint on the store instruction. A fresh build image initializes its own sample byte to zero.

The monitor still shows the house. At BASIC line 80, that is the expected picture. In Graphics Preview, inspect the hidden page at mapped $1A00 in PMODE 1, Set 0. Read X=81, Y=32 there. After restoring any edit, remove the line breakpoints, Continue in the BASIC workspace, click the CoCo display, and tap and release Space once.

A RAM address seems much too large. Check the heading above the number. The CPU has a 16-bit address. Physical RAM on this 512 KB machine extends to $7FFFF. A CPU address and a physical RAM address can denote the same byte at this stop. Record both alongside the active map before changing the machine.

Trace stopped, but the screen is still changing. A trace stop closes the recording. Use a breakpoint when you also need to hold RAM and the video page still. A trace sample stores CPU state at an earlier moment. Selecting it does not rewind RAM.

Books behind the experiments

- Tandy Corporation. *Color Computer 3 Service Manual*. Section V, "Theory of Operation," figure 5-7, "MMU Block Diagram." Use the diagram to follow the path from the CPU's address lines to the banked memory.
- Tepolt, Laurence A. *Assembly Language Programming for the CoCo 3*. TEPCO, 1987. Figures 3-4 and 4-1, pp. 20 and 23. Processor and video addressing on the CoCo 3.
- Motorola, Inc. *MC6809–MC6809E 8-Bit Microprocessor Programming Manual*. M6809PM (AD), March 1981, reprinted May 1983. Chapters 1–2 and Appendix A. Registers, condition codes and instruction behavior.
- Atkinson, Darren. *Motorola 6809 and Hitachi 6309 Programmer's Reference*. 2009. Parts I–II. Instructions and register differences between the two processors.

For assembly syntax, operands and condition codes, the Book 9 instruction directory is available offline. For the Mac controls around the machine, use The VCC Plus User's Handbook.

The Color Computer Archive preserves historical documentation that informed this book. The lessons and programs here were written for VCC Plus.

Original VCC Plus prose and instructional images © 2026 Brian K. Holman. See the application's **Licenses & Notices** and the library's artwork credits.

Publication and artwork notices

Original VCC Plus prose and instructional images © 2026 Brian K. Holman. Original component and example notices remain with their respective material.

Except where otherwise identified, original artwork owned by Brian K. Holman is © 2026 Brian K. Holman. All rights reserved.

Permission is granted to use and redistribute this artwork unchanged with VCC Plus, including modified versions of its GPL-licensed software. Other reproduction, modification or redistribution requires prior written permission, except as permitted by applicable law. Third-party materials remain subject to their respective licenses and rights.

The CoCo 3 and CM-8 monitor presentation in VCC Plus's skeuomorphic System view was inspired by SockMaster's "pic of CoCo 3 and CM-8 monitor," published on CoCopedia. VCC Plus's artwork was designed and rendered independently. The remaining hardware illustrations were independently designed using historical photographs and renderings as references.

Read this book online

CMOC license notice · LWTools COPYING notice