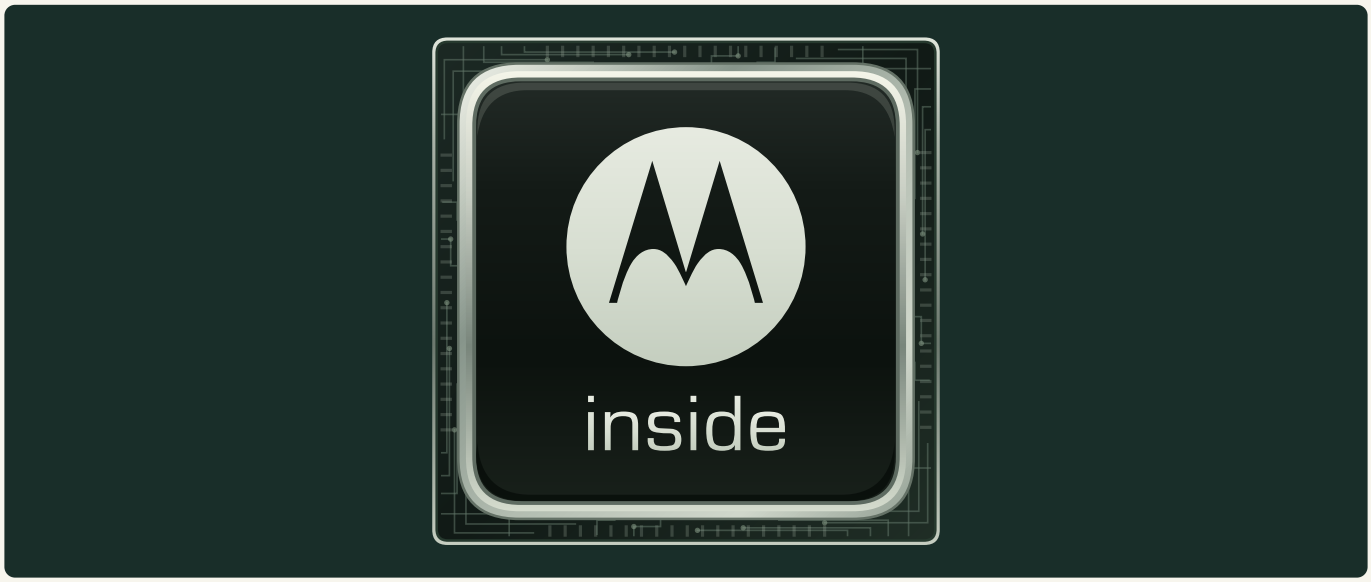
Your First 6809 Programs

A key goes down. A letter lights up. Stop the CoCo between those two moments and see what happens, one instruction at a time.



The VCC Plus Motorola processor badge with its silver frame and charcoal face.

Brian K. Holman

Book 7 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Where does one byte go? . . . 3
Send a letter to the screen . . . 6
Ask the CoCo to choose . . . 8
How does it find its way back? . . . 10
Fly to either planet . . . 13
At the assembly workbench . . . 17
Publication and artwork notices . . . 19

Where does one byte go?

Chapter 1 of 5

The CoCo is about to add one to 41. You can do the arithmetic in your head. Can you catch the instant the answer appears inside the machine? Our first program has just three instructions. You get to let them run one at a time.

Put the program on the workbench

1. Start with the **Disk BASIC** profile at the **OK** prompt. Save any guest program you want to keep. The exercise loads machine instructions into the CoCo's RAM.
2. Save byte.asm in a practice folder on your Mac. The file holds the words the programmer wrote. Choose **Develop → Assembly Development...**. Inside that window, choose **File → Open Source...** and open your saved file. Check that the document picker shows **byte.asm**. Your other open sources are available in the same picker.
3. Select the **MC6809** target and click **Assemble**. The assembler translates the words into bytes the processor can execute. Open **Build details → Listing** to see the words and bytes side by side. You can assemble a program while the CoCo is still sitting at its prompt.
4. Choose **Debug from Entry**. Confirm the load when asked. The CoCo now has your program and is paused at its first instruction. The footer should show **PC $3F00**. **PC** is the program counter. Its value is the address where the processor will fetch its next instruction.

```
            org     $3f00
start       lda     #$29        ; Load the number $29 (decimal 41) into A.
            inca                ; Add one. What will A hold after this step?
done        rts                 ; Return to BASIC with the result still in A.
            end     start
```

Read the left column as names for places. The label **start** names the address of the load. The label **done** names the address of the return. **ORG** tells the assembler where the program belongs, and **END start** tells it which label supplies the entry address. The three lines between them are the three instructions you will execute. Everything to the right of a semicolon is a note for the human reader.

Follow a byte through A

A is an accumulator, a small storage place inside the processor. It holds one byte, eight bits. In the first instruction, **LDA** means "load accumulator A." The **#** tells the assembler that the number following it is the value to load. The **$** tells it how to read the digits. In hexadecimal, each place counts sixteen of the place to its right. So **$29** is two sixteens plus nine, or **41**.

Click **Step** once. Read both **PC** and **A** in the footer. PC has moved to **$3F02** and A holds **$29**. Why did PC move by two? Find **86 29** in the listing. The first byte says which operation to perform. The second supplies its value. The processor has finished with both.

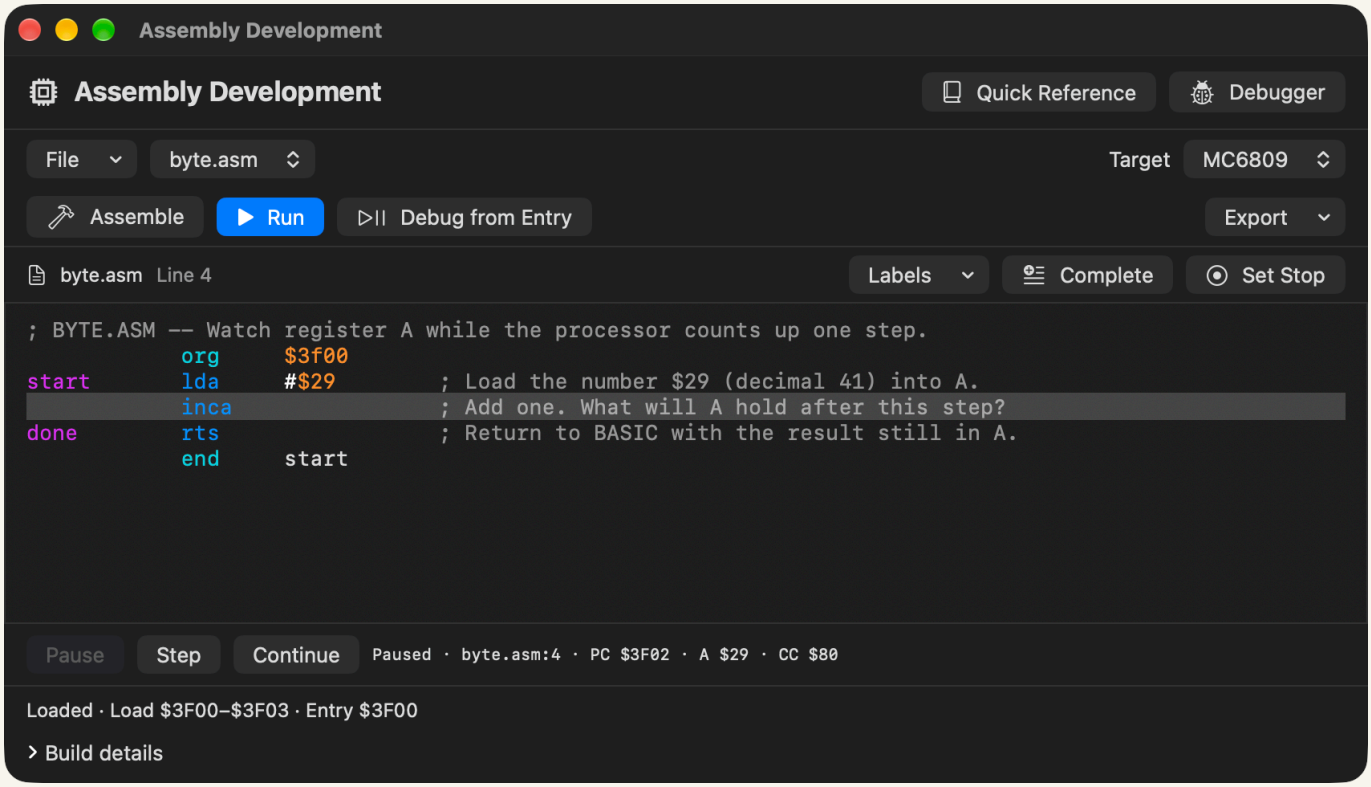


Figure 1: The assembly workspace pauses at INCA. The footer shows PC $3F02 and A $29 after executing the load.

The highlight is now on **INCA**, "increment A." Write down what you expect A to show after the next step. The digits in hexadecimal continue from 9 through A, B, C, D, E and F. What comes immediately after **$29**?

Click **Step** again. A reads **$2A**, or **42**. PC reads **$3F03**, the address of **RTS**. INCA occupied just one byte. The letter A in **$2A** is a hexadecimal digit worth ten. The A beside the value in the footer names the register holding it.

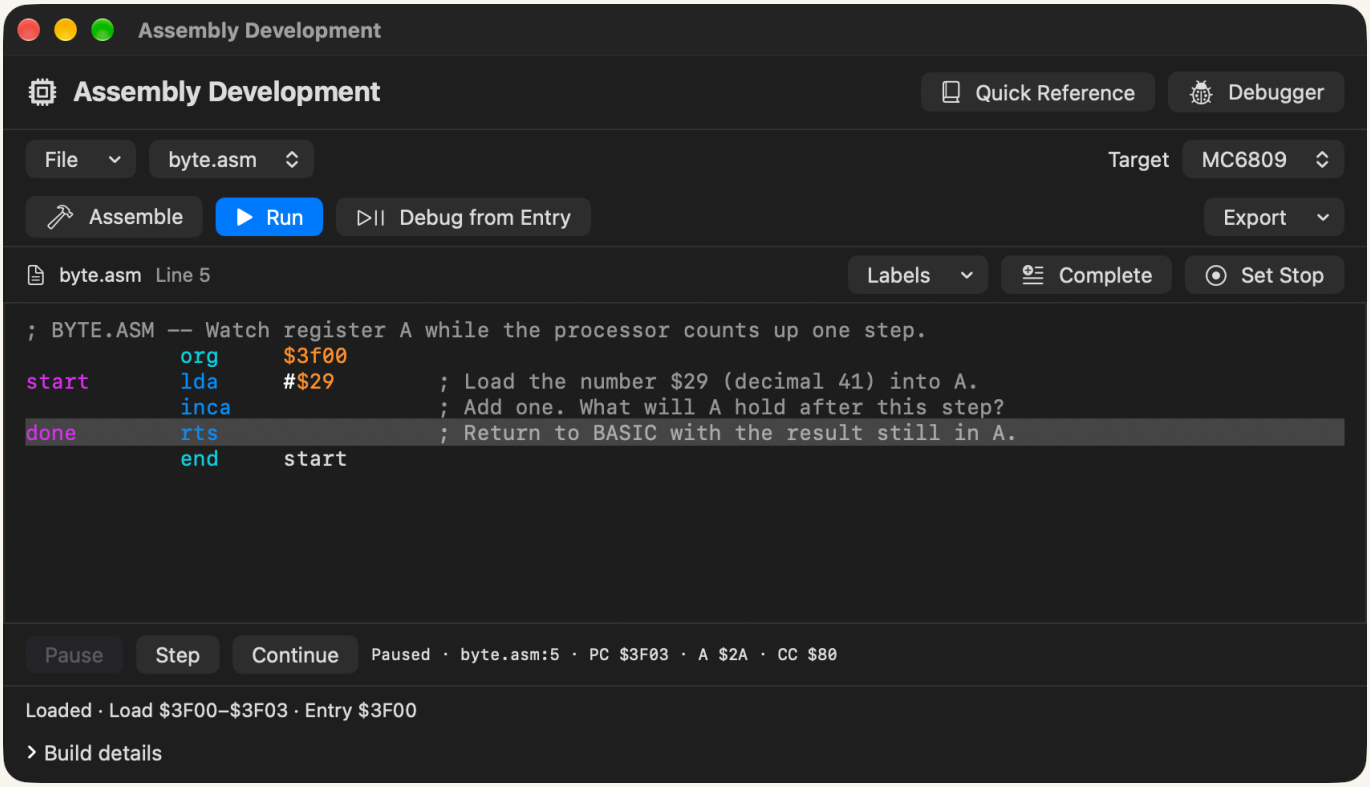


Figure 2: The next step has reached RTS. The footer shows PC $3F03 and A $2A, the result of INCA.

Let BASIC have its turn

Choose **Continue**. RTS returns control to the caller. VCC Plus started this program with a BASIC return address ready for it, so BASIC can now accept a command. Click the CoCo display, type **PRINT 2+3** and press **Return**. A fresh **5** and **OK** tell you who is listening.

Did you expect **42** to appear on the CoCo screen? The load and increment did all their work inside the processor. You saw their result because the Mac debugger let you look in A while the machine was paused. Starting with **Run** would have carried the CoCo straight through all three instructions. You can repeat the experiment with **Debug from Entry** to catch it again. In Send a letter to the screen, you will give the byte somewhere else to go.

If you came here from the BASIC loop that puts a number on screen, compare the size of a step. BASIC could make a whole statement its next action. Here one step took you only as far as loading a number into A.

Send a letter to the screen

Chapter 2 of 5

A register can hold a letter, provided the program and the screen agree on a code for it. The CoCo already knows how to print a character. Give its output routine one byte and watch where the letter appears.

Print one letter

Save letter.asm in your practice folder. In **Assembly Development**, choose **File → Open Source...** and open it. Keep the **MC6809** target. At the Disk BASIC prompt, choose **Run** in the assembly window and confirm the load if asked. The CoCo should print **A** on a line by itself and return to **OK**. This time the program did something you can see after it has finished.

```
            org     $3f00
start       lda     #'A         ; Load this character's ASCII code into A.
            jsr     [CHROUT]    ; Call the routine whose address is at CHROUT.
            lda     #CR
            jsr     [CHROUT]    ; Leave the next BASIC prompt on its own line.
            rts                 ; Return to the caller in BASIC.
            end     start
```

The character literal in the first LDA is written **#'A**. In this LWASM spelling, a single quote introduces the character that follows it. The assembler looks up its numeric character code. For uppercase A the code is **$41**, or decimal 65, in a code called ASCII. The instructions still deal in ordinary bytes. The meaning of the byte depends on what receives it.

The first **JSR** calls a subroutine. Its name stands for "jump to subroutine." Our subroutine is part of the Color Computer's built-in ROM, the read-only memory that also supplies BASIC. Its job here is to take the character code in A and send it to the current output, the screen in this exercise. When it finishes, execution resumes on the following line of our program.

The next load puts **CR** in A. Look near the start of the source for its definition. **EQU** gives a name to a value, and **CR** has the value 13. A carriage return is also a character code. The second call asks the ROM to start a new line. That's how the printed letter gets a line of its own.

Change the message with one keystroke

1. In the assembly window choose **File → Save a Copy...** and save **my-letter.asm**. Choose **File → Open Source...** and open that copy. Check the selected filename before you edit.
2. In the first LDA, change just the A following the quote to uppercase **Z**. Leave the A in **LDA** alone. Choose **File → Save Source**, then **Debug from Entry**. At the first instruction click **Step** once.
3. The footer should now show **PC $3F02** and **A $5A**. Which part of the assembled instruction changed when you chose a new letter? Open **Build details → Listing**. The two bytes in the original were **86 41**. The edited instruction uses **86 5A**.
4. Choose **Continue**. Look for **Z** on the CoCo. The same ROM routine just printed a different letter. If the output is still A, check which document is selected and compare its assembled bytes with the two above.

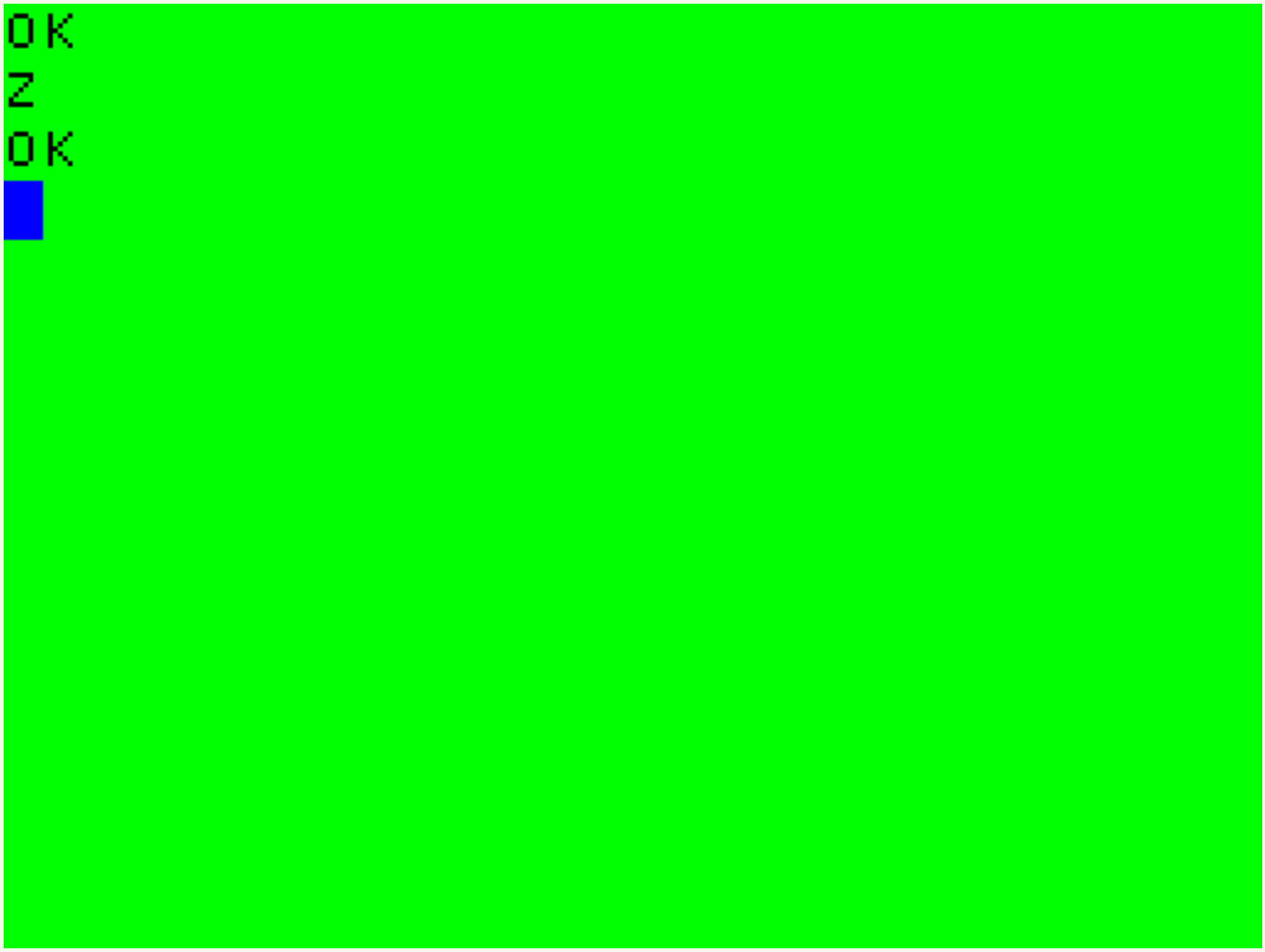


Figure 3: The CoCo screen shows Z on its own line, followed by the BASIC prompt, after running the edited character program.

Read the square brackets

```
CHROUT      equ     $a002       ; Address of the ROM character-output vector.
CR          equ     13          ; Carriage return starts a fresh line.
```

CHROUT names **$A002**. At that address the CoCo stores a two-byte address pointing to its character-output routine. An address used like this is called a **vector**. On **JSR [CHROUT]**, the processor first finds the address stored at CHROUT and then calls the routine there. The brackets tell the processor to take that extra look. The character itself travels in A.

If you **Step** while JSR is highlighted, the PC follows that call into the ROM. The assembly window can say **no verified source line** because the instructions at the new address belong to the CoCo's ROM. Choose **Continue** to finish. You'll soon write a subroutine of your own, where both the call and the code it reaches appear in the same source file.

You've sent a character *out* of the CoCo. Next you'll give a player a chance to send one back.

Ask the CoCo to choose

Chapter 3 of 5

A question mark waits on the CoCo screen. Tap X and it keeps waiting. Tap 2 and it answers M for Mars. There are only two accepted keys, but the keyboard can supply many more. Where does the program send all the others?

Try the two routes

Save choice.asm and open it with **File → Open Source…** in **Assembly Development**. From the Disk BASIC prompt choose **Run**. Click the CoCo display. When **?** appears, press and release **X**. Give the program a moment, then press **2**. A single keypress is enough. Look for **?M** and **OK**. Run again and press **1**. This time look for **?E**.

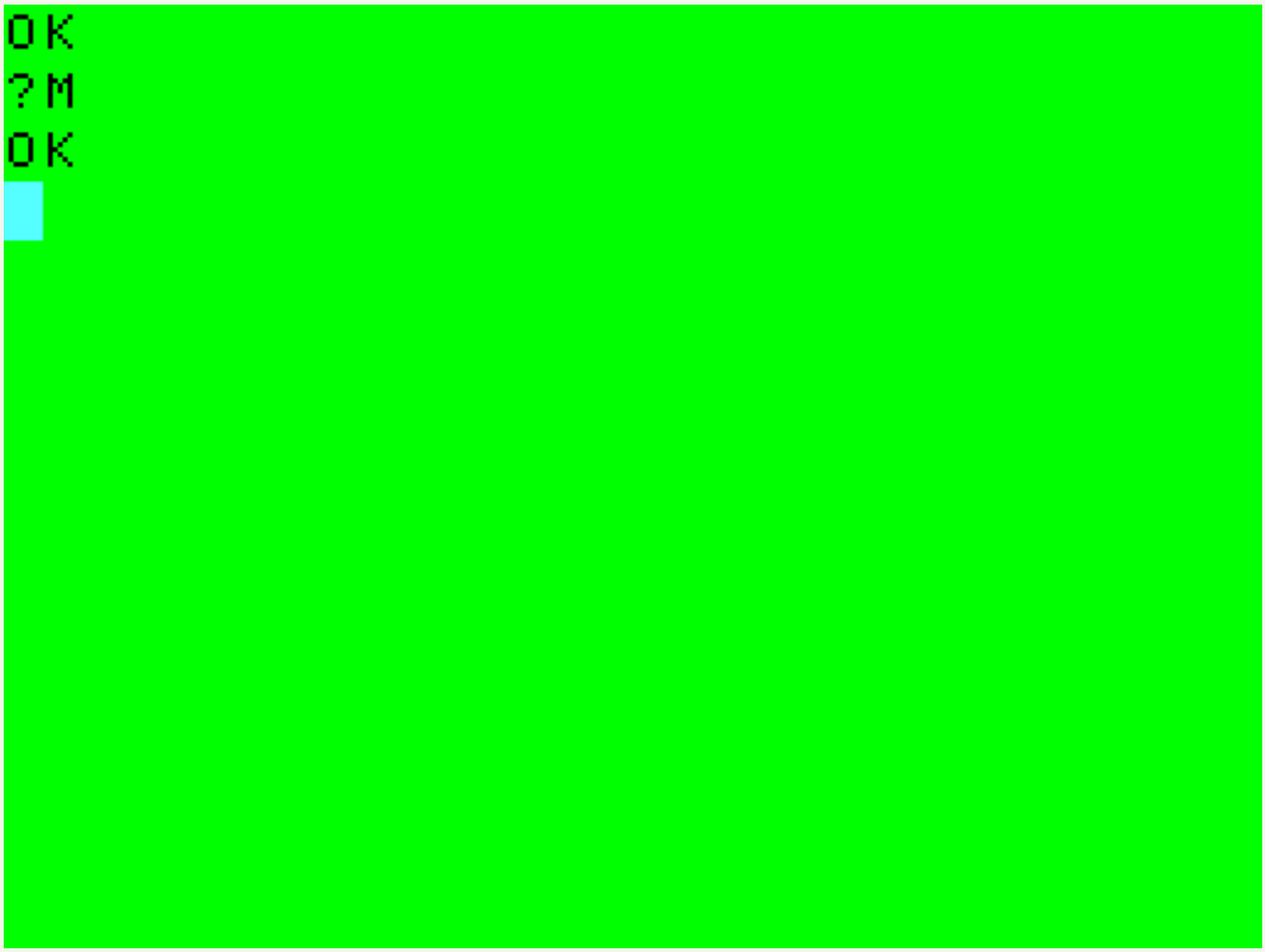


Figure 4: The CoCo display shows ?M and returns to BASIC after the Mars key was accepted.

If nothing happens when you press 2, look for the blinking insertion point in the Mac editor. Choose **Input → Focus CoCo Display** to send keystrokes to the guest instead. The program polls the CoCo keyboard itself. It has no need for a Return key to finish a line of typing.

Read the question from the keyboard

```
start       lda     #'?         ; Show that the program is waiting for a key.
            jsr     [CHROUT]
choose      jsr     [POLCAT]    ; Check the keyboard without waiting for Enter.
            cmpa    #'1
            beq     earth       ; A matching comparison sets the Z flag.
            cmpa    #'2
            bne     choose      ; No key or an unexpected key: try again.
```

The first two instructions print the question mark using the output vector from the previous chapter. **POLCAT** is another ROM vector, this time for checking the keyboard. The call at the label **choose** returns a character code in A, or zero when there is no key to report.

CMPA compares the value already in A with the value in its operand. Imagine subtracting the two to see whether the answer would be zero. The comparison leaves A alone and records the result in the **condition-code register**, abbreviated **CC**. One bit in CC is the **zero flag**, **Z**. Equal values give a difference of zero and set that flag. The following branch can use it to choose the next instruction.

Start at **choose** and take the Mars route by hand. When the player presses 2, A holds **$32**, the ASCII code for the key marked 2. The first comparison asks if it matches the code for the 1 key, **$31**. **BEQ** means “branch if equal,” so that branch passes by. The second comparison finds a match. **BNE** means “branch if unequal.” This time it passes by too. The next instruction in memory happens to be the one at **mars**. A branch can make a choice by letting execution carry on in order.

Now try the same walk for X. It differs from both accepted character codes. Which instruction sends the processor back to **choose**? Zero from an idle keyboard follows that same route. The question mark gets printed only once because the loop goes back to the keyboard call, below the print.

Catch the choice before it prints

```
mars        lda     #'M
            bra     printit     ; Skip the Earth path.
earth       lda     #'E
printit     jsr     [CHROUT]    ; Both paths meet here with one letter in A.
            lda     #CR
            jsr     [CHROUT]
            rts
```

1. Click anywhere on the line that starts **mars** and choose **Set Stop**. Do the same on the line beginning **earth**. The **Stops** menu should now list two locations. Both stops are on instructions that will execute only after the program accepts a key.
2. Choose **Run**. At the question mark press **1**. The workspace stops before the Earth load at **PC $3F16**. The incoming character is still in A, so expect **$31**. **Step** once. The load has replaced the player’s key with **$45**, the code for uppercase E. **Continue** to let the guest print it.
3. **Run** again and press **2**. At the Mars stop PC is **$3F12** and A is **$32**. Predict what the screen will show. **Continue** to check. The **BRA printit** below the Mars load always branches. It carries the M past the Earth load so both routes can use one output call.
4. Choose **Stops → Clear All Stops** in the assembly workspace. A fresh run can now play straight through. To look up one of the decisions later, open **Develop → Assembly Quick Reference…** and search for **CMPA**, **CC.Z**, **BEQ** or **BNE**.

Compare this pair of decisions with the BASIC planet menu or the C program’s key loop. All three ask the same question. In assembly you can point separately to the instruction that compares two values and the instruction that acts on its answer.

How does it find its way back?

Chapter 4 of 5

Put a finger on this sentence, glance at the next heading, then come back to your finger. You just kept your place through a detour. The CoCo has to do that every time one piece of a program asks another to do a job. In this experiment you can read the two bytes it uses to keep its place.

Write down the address you expect

Save return.asm and open your copy with **File → Open Source...** in Assembly Development. Start from Disk BASIC with the **MC6809** target. Clear any stops in this document. Look at the caller and helper together.

```
            org     $3f00
start       lda     #'A         ; Begin with the letter A ($41).
call        jsr     nextletter  ; Save the address of 'display' on the S stack.
display     jsr     [CHROUT]    ; The subroutine has left the new letter in A.
            lda     #CR
            jsr     [CHROUT]
            rts                 ; Finish the main program and return to BASIC.

nextletter  inca                ; ASCII B follows ASCII A.
back        rts                 ; Retrieve the saved address and resume at display.
            end     start
```

The caller puts the code for the letter A in register A. The helper, **nextletter**, adds one. The agreement between these two pieces of code is simple: the caller puts a character code in A and the helper leaves the next code there. Both pieces can read and change that same register.

Choose **Debug from Entry** and confirm. **Step** once. The footer should now read **PC $3F02 · A $41**. Open **Build details → Listing**. Find the JSR instruction at **call**. It occupies three bytes starting at $3F02. The instruction marked **display** starts immediately afterward. Which address will the processor need when the helper has done its job? Write it down before stepping.

Read what the call saved

Open **Debugger → Inspector** from the workspace. Read **S**, the system stack pointer. The stack is a region of RAM the processor uses to keep things it will need back, most recently saved first. S gives the address at its top. Keep the Inspector handy and use **Step** in Assembly Development.

When you look	Where PC points	What to check
Before JSR	**call**, $3F02	Write down S. A contains $41.
After one Step	**nextletter**, $3F10	A still holds $41. S is two less than the value you wrote down.
After the next Step	**back**, $3F11	INCA has produced $42. The helper's RTS is next.
After stepping that RTS	**display**, $3F05	S is back to its starting value. A still holds $42.

Pause on the **second row** before going further. In the Inspector's lower **Memory** pane, choose **Mapped CPU**. In **Address** enter the current S in hexadecimal and choose **Go**. The debugger displays memory in rows of sixteen bytes. For S=$7F33, find row **7F30**, column **03**, then move one column right to **04**. For another S, use its last hexadecimal digit to find the column in the same way. With this source you will find **3F 05**. Together they spell **$3F05**, the address of **display**. The CPU first made room for two bytes by moving down the stack. Those two bytes can describe any address in its 16-bit address space.

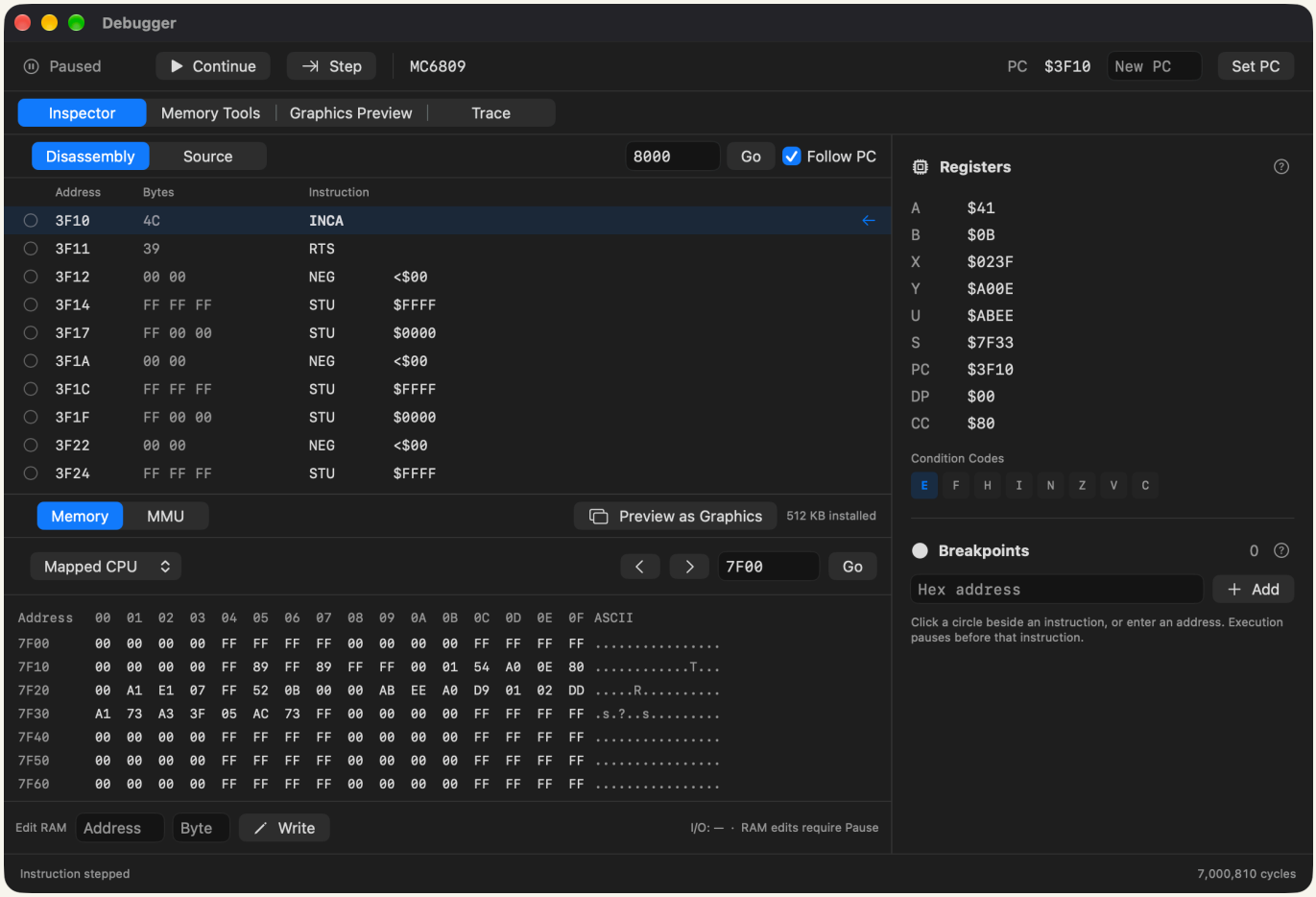


Figure 5: The debugger shows the saved return address in guest RAM during the call to nextletter. Read the two bytes at S in order, 3F then 05.

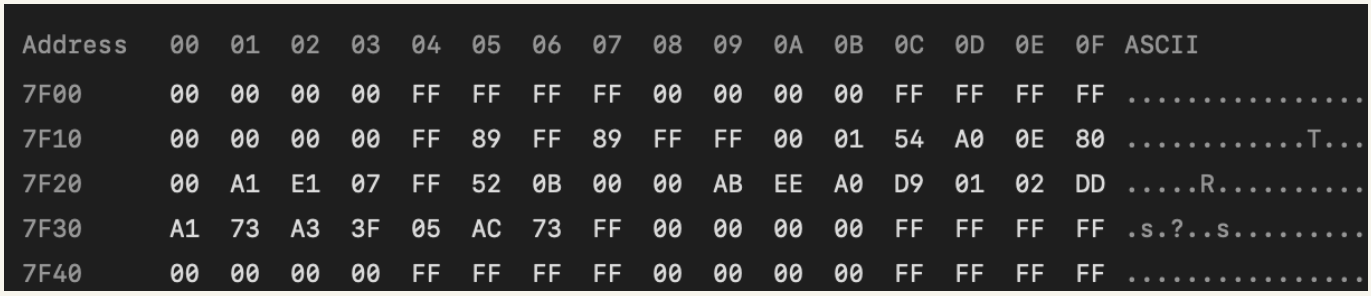


Figure 6: In the magnified memory grid, row 7F30 contains 3F under column 03 and 05 under column 04.

If your S differs from the figure, use the value on your own machine. The difference of two and the contents at that address tell you what JSR did. Inspect the bytes without writing to them. The processor will soon rely on their contents.

One return to the caller, one return to BASIC

Finish the last two steps in the table. At **display**, the helper has returned its answer in register A. The CoCo has yet to print the letter. Choose **Continue**. The call through CHROUT hands $42 to the ROM, which prints **B**. The following carriage return starts a fresh line. Finally the main routine reaches its own RTS and BASIC displays **OK**.

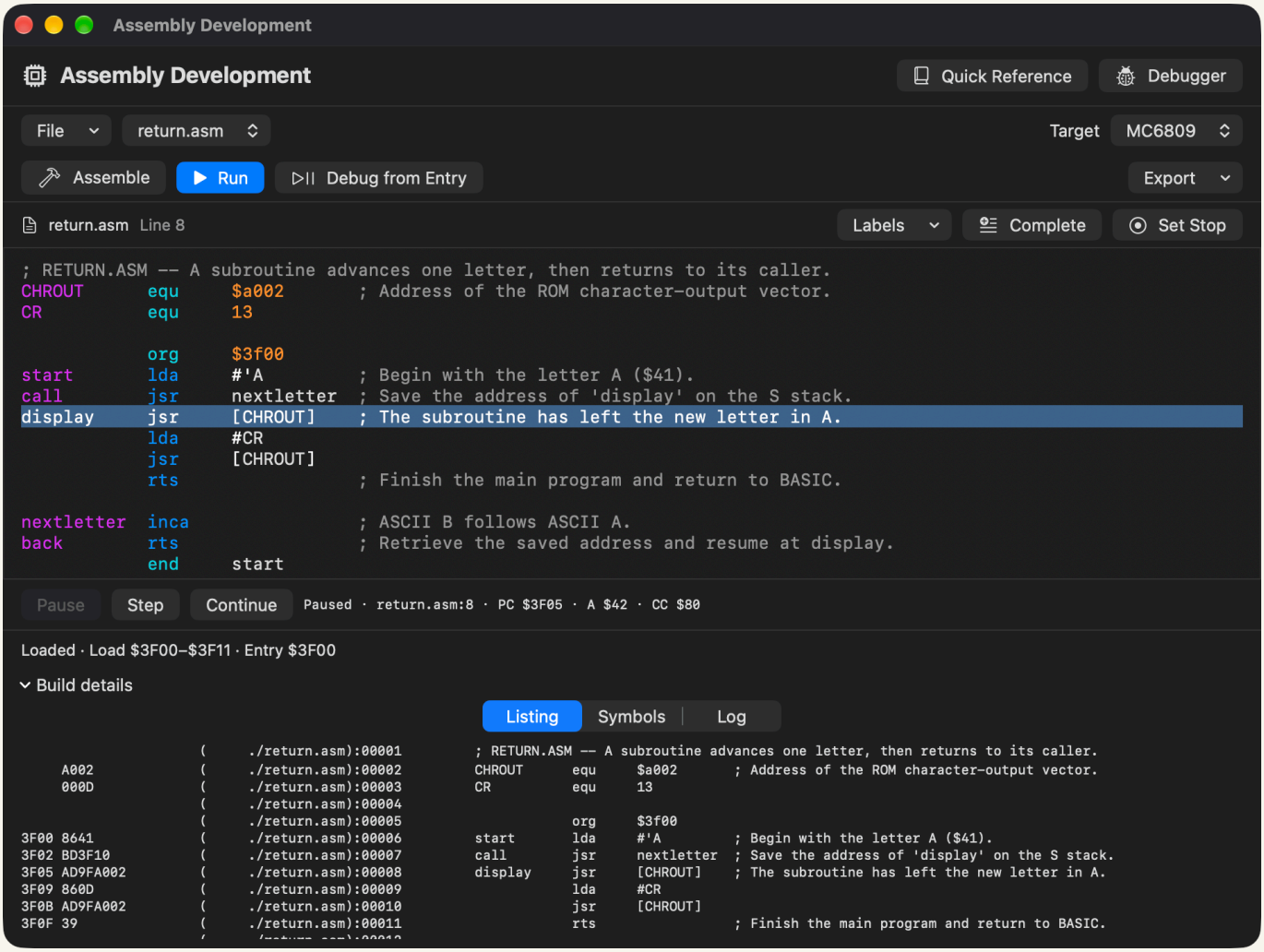


Figure 7: After the helper returns, display is highlighted with PC $3F05 and A $42. The listing below shows the call at $3F02 followed by display at $3F05.

Now find the two RTS instructions again. At **back**, the top of the stack holds the address the JSR at **call** saved. At the main routine's final RTS, the helper's address has already been used. The BASIC-hosted launch supplied a caller for the whole program. The same one-byte instruction can therefore take you to two entirely different places. The top of the stack decides where you go.

If you've used the C call stack, you have seen a caller named for you in its own window. Here you found the caller's next instruction using only a listing, S, and two bytes of RAM. Next, you can use that same idea when Earth or Mars asks a helper to print a whole string.

Fly to either planet

Chapter 5 of 5

Someone playing Earth or Mars needs just one finger. Someone reading the assembly can answer a harder question: where in the program has the CoCo already chosen the greeting, even though the player can’t see it yet?

Take both flights

Start at a Disk BASIC prompt. In **Develop → Assembly Development…**, use the workspace’s **File → Hello World — Earth or Mars**, or save planet.asm and use **File → Open Source…**. Select the **MC6809** target. Choose **Run** and confirm the load if asked. Click the emulator display. When the menu asks which planet you live on, press **1** without Return. Run again and press **2**. Try **3** before the valid key on one flight. The menu keeps waiting.

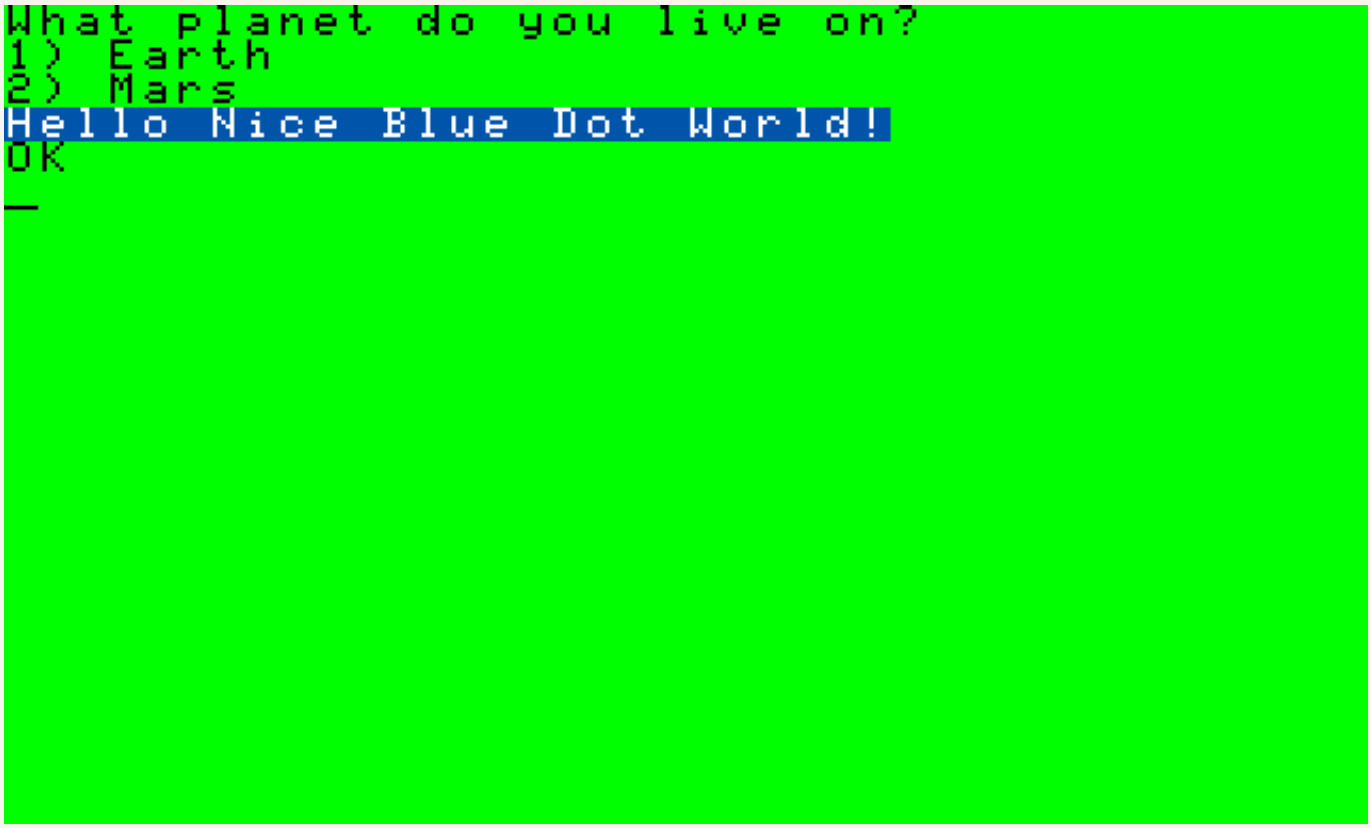


Figure 8: The CoCo 3 shows the two planet choices, then Hello Nice Blue Dot World! on blue after key 1.



Figure 9: The same CoCo 3 display after key 2 shows Hello Cold Red Dot World! on red. BASIC is ready for another command.

The starter opens a **40-column** screen. The first two instructions put decimal 40 in B and call the CoCo 3 ROM routine labeled WIDTH. You can spot decimal 40 as hexadecimal $28 after that first step. The lowercase letters in “Hello” and the separately colored greeting use the CoCo 3 text display. Beneath each greeting the ordinary BASIC prompt is available again.

Catch the instant both routes meet

In the source, find **greet** at the instruction **STB BG6**. Put the caret there and choose **Set Stop**. Use **Run** again, return focus to the CoCo and press a planet key. The editor stops before the store. Both paths have already done two jobs. They chose a color byte in B and the address of a message in X.

```
mars        ldb     #$25        ; Warm red background, visible on RGB and CMP.
            ldx     #redmsg
            bra     greet
earth       ldb     #$0a        ; Cool blue background.
            ldx     #bluemsg

greet       stb     BG6         ; Choose the tint for the greeting only.
            lda     #%00011110  ; White foreground (3), background slot 6.
            sta     ATTR
            bsr     puts        ; X points to the selected greeting.
```

Open **Debugger → Inspector** to read B and X. Check **Build details → Symbols** in the assembly editor for the address beside **bluemsg** or **redmsg**. Only after taking your own readings, compare them with these:

Key	B at greet	X at greet in the supplied starter	Prediction
1	$0A	$3F73, bluemsg	“Hello Nice Blue Dot World!” on blue

Key	B at greet	X at greet in the supplied starter	Prediction
2	$25	$3F8E, redmsg	“Hello Cold Red Dot World!” on red

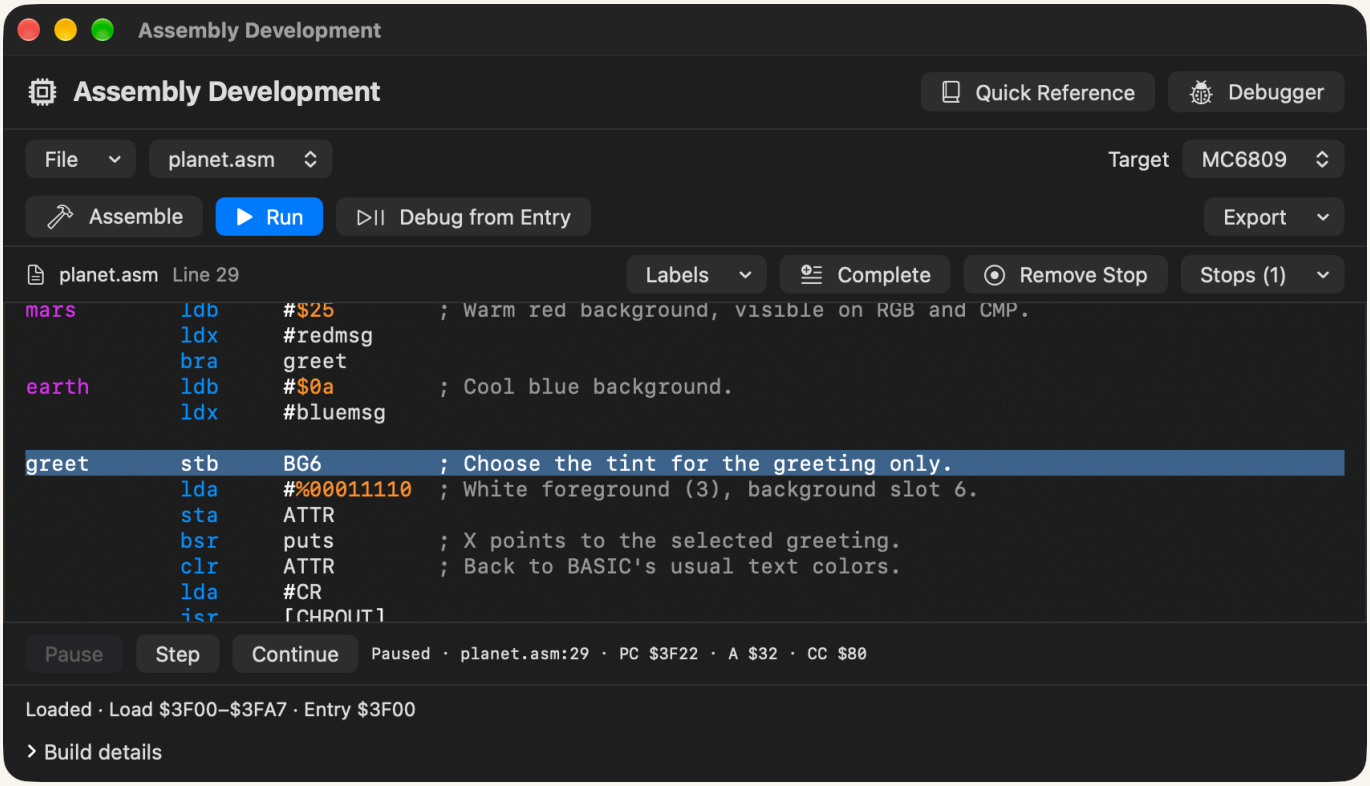


Figure 10: The assembly debugger stopped at greet. The next instruction will store register B in the background palette slot.

Why does Earth need no branch after it sets X? Read the instruction immediately below the Earth block. It is already **greet**. Mars must branch over the Earth block to reach the same spot with its own choices intact. The short keyboard exercise used this same arrangement with one letter.

Step once. STB took the color in B and wrote it to $FFB6, the CoCo 3 GIME’s palette slot 6. In this text mode, a character’s attribute selects a palette slot for the background. The following instructions set the text attribute to use slot 6 for the greeting. One action chooses the paint in the pot and the other chooses which pot to use. The greeting itself still has to be printed. Choose **Continue** to let that happen. Remove the stop when you have tried both predictions.

How X gets a sentence onto the screen

An address fits in X, which is sixteen bits wide. Each letter fits in one byte. At **bluemsg** or **redmsg**, FCC lays the letters next to each other in memory and FCB places a zero byte after the last one. The two messages can be different lengths because the zero marks where to stop.

```
; Print the zero-terminated string at X. X+ advances to the next byte.
puts        lda     ,x+
            beq     done
            pshs    x           ; Keep our place across the ROM call.
            jsr     [CHROUT]
            puls    x
            bra     puts
done        rts                 ; Return to the caller of puts.
```

Read **LDA ,X+** aloud: “Get the byte at the address in X, then advance X to the next byte.” Loading zero sets the zero flag, so **BEQ done** takes the return route when it reaches the terminator. For an ordinary letter the program saves X on the stack, calls CHROUT with the letter in A, and restores X to find the next one. This is the same output routine used to print a single letter. Here the caller also keeps its place in the sentence. The menu and both greetings can share this one little printing loop.

Land in BASIC

Once the helper has printed the greeting, the caller clears the text attribute for subsequent output, prints a carriage return and executes RTS. The 40-column width persists. Click the CoCo screen, type **PRINT 2+3** and press **Return**. Seeing **5** tells you that BASIC is accepting a fresh command in the same running machine.

Think back to the decision you made at the keyboard. In BASIC you can follow the answer to a numbered line. In C you can follow the key into a condition. In assembly you can stop with the key still in A, or a few instructions later with a color in B and a message address in X. Pick the view that answers the question you have.

The field guide collects the five editable programs, a short guide to the registers and return paths, and the books behind the lessons.

At the assembly workbench

Appendix A

Five small files can tell you a surprising amount about a computer. Choose one by the question you want to answer. Each link gives you editable text in the same form the bundled LWASM assembler reads.

Save a program of your own

I want to see…	Take this source	Where to stop
One byte change	byte.asm	Stop at entry, then follow A through two instructions.
One byte appear as a letter	letter.asm	Stop just before CHROUT, then compare the byte with the character.
A key choose a path	choice.asm	Stop on the first load in either planet branch.
A subroutine remember its caller	return.asm	Step into nextletter, then read the return address at S.
The whole 40-column planet game	planet.asm	Stop before the palette write and predict the greeting from B and X.

In **Help → Manual Library…**, select a source link and choose a location in **Save Assembly Example**. In the standalone browser edition, use the download or take the file from the book's **examples** folder. From **Develop → Assembly Development…** choose **File → Open Source…**. Opening a document gives you a copy to study. **Assemble** translates it. **Run** or **Debug from Entry** loads the resulting machine code into the guest. Before a load, save any guest work you want to keep.

Use the workspace's document picker to go back to an earlier source. If you change an instruction and want to observe the change, save and run the new build. A breakpoint applies to the instructions from the matching build. Paper notes about an address can grow stale when a program changes size.

Read a stopped machine

What the debugger shows	How to use it in this book
PC · program counter	Find this address in **Build details → Listing**. The source highlight and PC identify the instruction about to execute. Record changes after **Step**.
A, **B** · 8-bit accumulators	Each holds one byte. The program decides what a byte means. In the planet example A can carry a key, while B can carry a color.
X · 16-bit index register	Look up its value under **Symbols**. A message label gives you a name for the address. In the printing loop X advances a byte at a time.
CC.Z · zero flag inside CC	Immediately after CMPA, equality sets Z. BEQ follows its branch when Z is set. Later instructions can change it.
S · system stack pointer	Note S before JSR and again inside the routine. Read two bytes in **Inspector → Memory → Mapped CPU** and put the high byte first. After the matching RTS, check both S and PC.

The assembly window's **Quick Reference** can look up the instruction at the caret. Open **Develop → Assembly Quick Reference…** for the full offline index. Try **Immediate** for the # sign, **CC.Z** for a branch decision, **Indexed** for ,X+, and **BSR** to compare a relative subroutine call with JSR. BSR and JSR both arrange for RTS to return to the following instruction.

Get back to the experiment

- **The program finished before you saw a register.** Choose **Debug from Entry**, then **Step**. Run is fast enough to finish the smallest examples before you can click Pause.
- **A key went into the editor.** Click the emulator display or choose **Input → Focus CoCo Display**. The planet choices take one key. A command at the BASIC prompt takes Return.
- **Step left the source file.** A JSR through CHROUT or POLCAT enters the ROM. Give the processor an instruction in your own file to stop on, then **Continue**. If you simply want the program to finish, clear the stops and continue.
- **A label has a value, but Set Stop cannot use it.** Use a line that assembles to an instruction, such as **greet STB BG6**. EQU assigns a name a value. FCC and FCB can lay down message bytes. These have different jobs from executable instructions.
- **You want to carry a finished program on disk.** The handbook's Assembly workspace tour covers Export Binary, Export Disk Image and optional DOS boot. Direct Run is enough for all five experiments here.

Ready to follow a message through several routines? Catch a three-byte packet in memory, or follow a saved sum onto the stack in the advanced assembly volume.

Books behind the workbench

- Motorola, Inc. *MC6809–MC6809E 8-Bit Microprocessor Programming Manual*. M6809PM (AD), March 1981, reprinted May 1983. Chapters 1–2 and Appendix A. For the registers, flag effects, addressing and instruction definitions used in the debugger experiments.
- Barden, William, Jr. *TRS-80 Color Computer Assembly Language Programming*. Radio Shack, 1983. For assembly practice on the Color Computer, including addressing and subroutines.
- Tepolt, Laurence A. *Assembly Language Programming for the CoCo 3*. TEPCO, 1987. Pages 20 and 23, figures 3-4 and 4-1. For further reading on the memory a program addresses and the memory the CoCo 3 uses for its display.
- Astle, William, and LWTools contributors. *LW Tool Chain*. Manual accompanying LWTools 4.25, chapter 3, "LWASM." For assembler source syntax, symbols and directives.

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. The cover shows the app's Motorola processor badge. The processor mark belongs to its owner. The app's **Licenses & Notices** identifies the emulator, assembler and firmware notices. 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