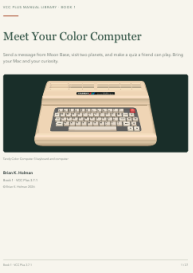
Choose a volume

Select a cover or title to jump to that book. Each book's own contents links to its chapters.



BOOK 1

Meet Your Color Computer

Open volume / PDF page 3



BOOK 2

Back at the CoCo

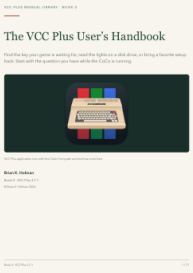
Open volume / PDF page 30



BOOK 3

Programming in CoCo BASIC on the Color Computer 3

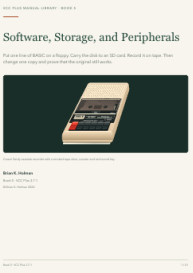
Open volume / PDF page 55



BOOK 4

The VCC Plus User's Handbook

Open volume / PDF page 134



BOOK 5

Software, Storage, and Peripherals

Open volume / PDF page 211



BOOK 6

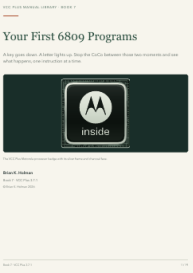
Programming in C on the CoCo 3 — CMOC and Its APIs

Open volume / PDF page 252

© Brian K. Holman 2026 / vccplus.brianholman.com

Choose a volume

Select a cover or title to jump to that book. Each book's own contents links to its chapters.



BOOK 7

Your First 6809 Programs

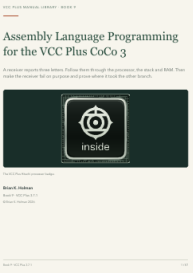
Open volume / PDF page 450



BOOK 8

NitrOS-9 on VCC Plus — A Practical Quick Start

Open volume / PDF page 469



BOOK 9

Assembly Language Programming for the VCC Plus CoCo 3

Open volume / PDF page 492



BOOK 10

Inside the CoCo 3 — Hardware and Debugging

Open volume / PDF page 579



BOOK 11

Working with OS-9 — A Comprehensive Manual

Open volume / PDF page 606



BOOK 12

Vibe Coding with ChatGPT

Open volume / PDF page 657

© Brian K. Holman 2026 / vccplus.brianholman.com

Meet Your Color Computer

Send a message from Moon Base, visit two planets, and make a quiz a friend can play. Bring your Mac and your curiosity.



Tandy Color Computer 3 keyboard and computer

Brian K. Holman

Book 1 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Meet the machine 3
Give one instruction 7
Send a message from Moon Base 9
Pick a planet 11
Send tomorrow a message 14
Catch the computer in the act 17
Give your friend the controls 21
Your field guide 24
On the bookshelf 26
Publication and artwork notices 27

Meet the machine

Chapter 1 of 7

Picture a family in 1986 making room for a computer. Someone clears the table in front of the television. Someone else sets down a cream-colored keyboard. Where does the computer go? It's already there, inside the keyboard case.

Find the keyboard and monitor

Tandy's Color Computer 3 arrived in 1986. It was the newest member of a family of computers that owners called the **CoCo**. Switch it on and you could give it an instruction in a language called **BASIC**.



Figure 1: Illustration of the Color Computer 3. The keys are built into the case that houses the computer.

A family could use its television for the picture or give the CoCo a monitor of its own. A separate drive could sit beside the computer. Slip a floppy disk into that drive and you had a place to keep programs for another day.

The original CoCo 3 had **128 KB of RAM**. RAM is the computer's working memory. KB stands for kilobytes, a way to measure how much it can hold. Owners could expand that space to **512 KB**, four times as much. Look for that number on the machine in this book.

Remember using that floppy drive? Find its controls on the Mac in *Back at the CoCo*.

Set a CoCo on your Mac desk

Open VCC Plus on your Mac. It's an **emulator**. Your Mac does the work of running a CoCo, right down to accepting the keys you press. For these lessons, start with the **Disk BASIC** setup and its supplied **Hello World** disk in **FD0**, the first floppy drive.

On a fresh setup, go straight to the main emulator window. If **Using your CoCo** is open, read the short guide, then click outside it to get back to your computer.

Already use VCC Plus? Make a practice setup

Save any CoCo work you want to keep before loading a setup. Loading one clears the CoCo's working memory. Here's how to give your book experiments a starter disk of their own.

1. Open **Machine → Configuration Profiles...**.
2. Choose **Disk BASIC** under **System Profiles**.
3. Choose **Create Fresh Starter Media...**.
4. Name your setup **My First CoCo** and choose **Save**.
5. With your new setup selected, choose **Load...**, then **Load Profile**.
6. Return to the main window.

If the machine is paused, choose **Run**.

Choose **Display → Interactive System**. Find the Color Computer 3 nameplate and the RAM badge. The case underneath the keys houses the computer. Look above the keys for the monitor, then beside the monitor for the disk drives. Find **FD0**. If you'd like a list of the drives, open **Media → Floppy Disks...**. FD0 should show **Hello World.dsk**.

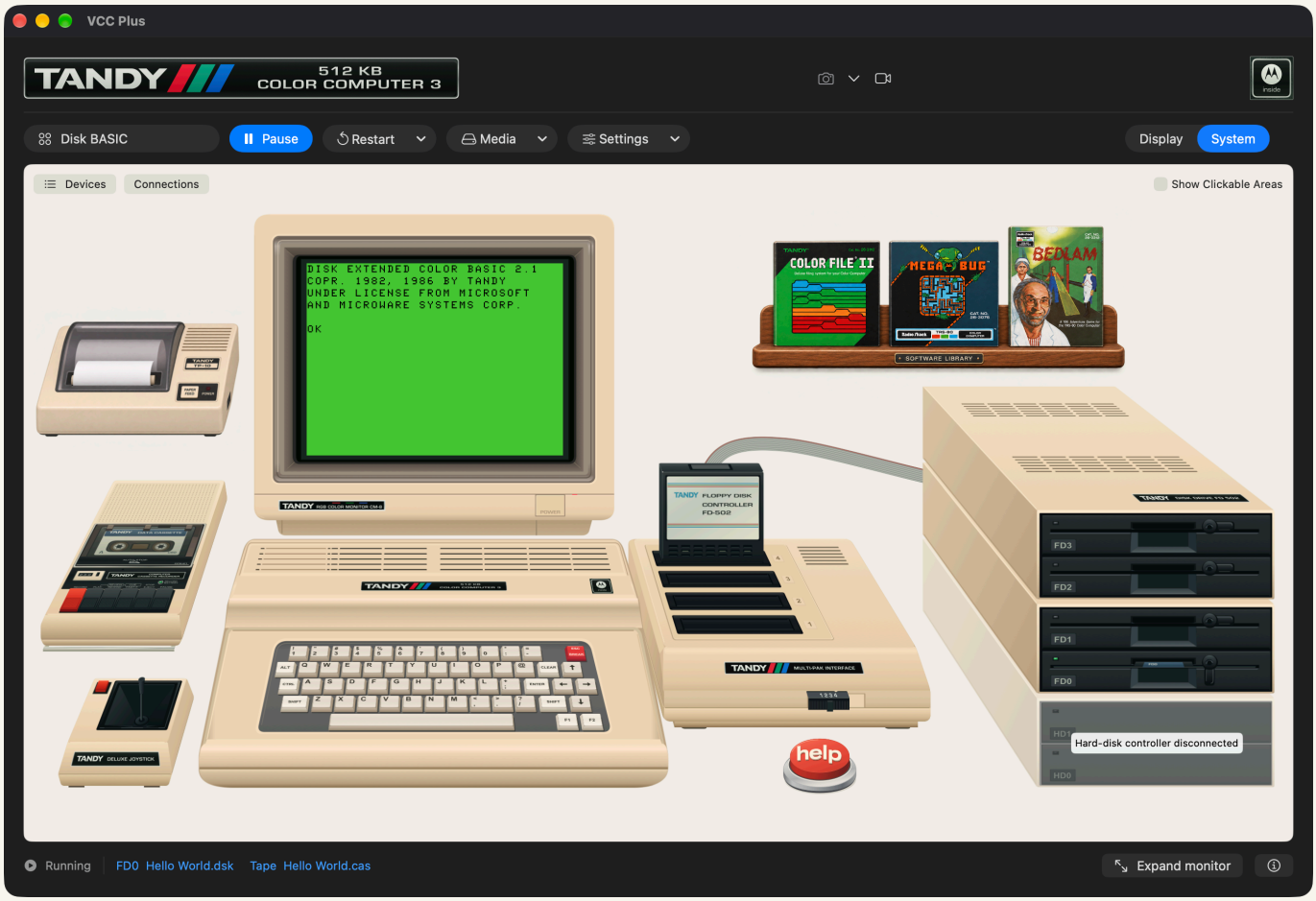


Figure 2: VCC Plus System view. Find the CoCo keyboard case below the monitor and the disk drives beside it.

Try pointing to three places in the picture. **Below the monitor**, find the keyboard where a person at a real CoCo would type. **Inside the monitor**, find the green screen where the answer appears. **To the right**, find the drive labeled FD0. A real floppy drive had a slot for a thin disk. Later you'll slip a whole disk into this one by choosing a file on your Mac.

Get a reply

Look for **OK** on the CoCo screen. That's the **prompt**. BASIC is ready for an instruction. The blinking square shows where the next character will appear.

1. Click inside the CoCo screen. From the keyboard, choose **Input → Focus CoCo Display** or press **Option-Command-K**. This directs your typing to the CoCo.
2. Type the line below, including both double quotation marks. Pause with your finger over Return.

```
PRINT "HELLO, COCO!"
```

So far, every letter on that line came from your fingers. Press **Return**. Look just below the line you typed.

```
HELLO, COCO!
OK
```

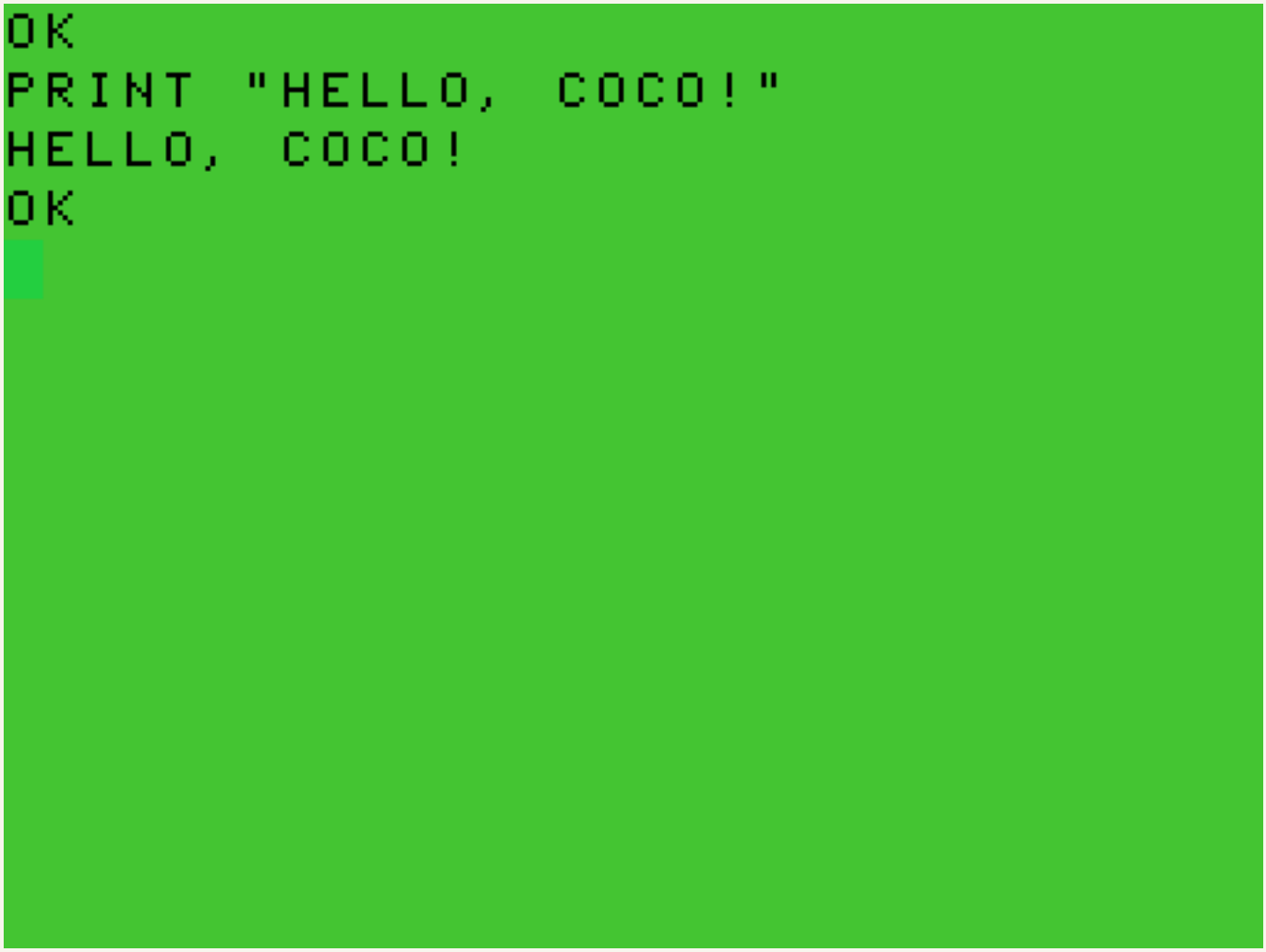


Figure 3: VCC Plus showing the typed PRINT instruction, the greeting on its own line, and the next OK prompt.

Your keystrokes were **input**. The CoCo followed your instruction and sent the greeting to the monitor. That greeting was **output**. Find the exclamation mark twice on your screen. Can you tell which one belongs to the instruction and which one belongs to the reply?

If typing goes to a Mac control, press **Option-Command-K** and try the line again. To go the other way, choose **Input → Focus Mac Controls** or press **Option-Command-U**. **Tab** moves between Mac controls and **Space**

activates a focused button. If Tab skips buttons, turn on **Keyboard navigation** in **System Settings → Keyboard** on your Mac.

Try leaving an answer half typed, then bringing the same window back in the handbook's flight-number exercise.

Where to go next

You supplied every letter of the greeting. Can you get the CoCo to print an answer you never typed? Leave it at the prompt for chapter 2.

Tandy Corporation. *Introducing Your Color Computer 3*. 1986. "To Our Customers" and the opening hardware tour.

CoCopedia contributors. "Color Computer 3." *CoCopedia*, revision 11134.

Give one instruction

Chapter 2 of 7

You've asked BASIC to show words you supplied. Now give it a plus sign. Can you make the sign disappear in one answer and show up in the next?

Try it on your CoCo

At the **OK** prompt, click the CoCo screen or press **Option-Command-K**. Type this line. Guess what will appear beneath it, then press **Return**.

```
PRINT 2+3
```

Check your prediction

The answer is **5**. Look closely and you'll see a space before the digit. BASIC leaves room for the sign when it prints a positive number.

Here are the same numbers and the same plus sign. This time, type a double quotation mark on each side of them. Make your prediction, then press **Return**.

```
PRINT "2+3"
```

You should see **2+3**. Inside the quotes, each character is a piece of text to show. The plus sign gets a spot on the screen along with the two digits. The quote marks themselves have done their job by telling BASIC where the text begins and ends.

Put the question and answer on one line

There are **two** plus signs in this next instruction. Point to the one you think will reach the screen. Type the line at the prompt and press **Return**.

```
PRINT "2+3=";2+3
```

```
2+3= 5
```

Read it from left to right. First comes the text between the quotes, right through the equals sign. The semicolon tells PRINT to carry straight on with the next item. Outside the quotes, the second plus sign tells BASIC to add. The result is a number, so it gets that space in front of it.

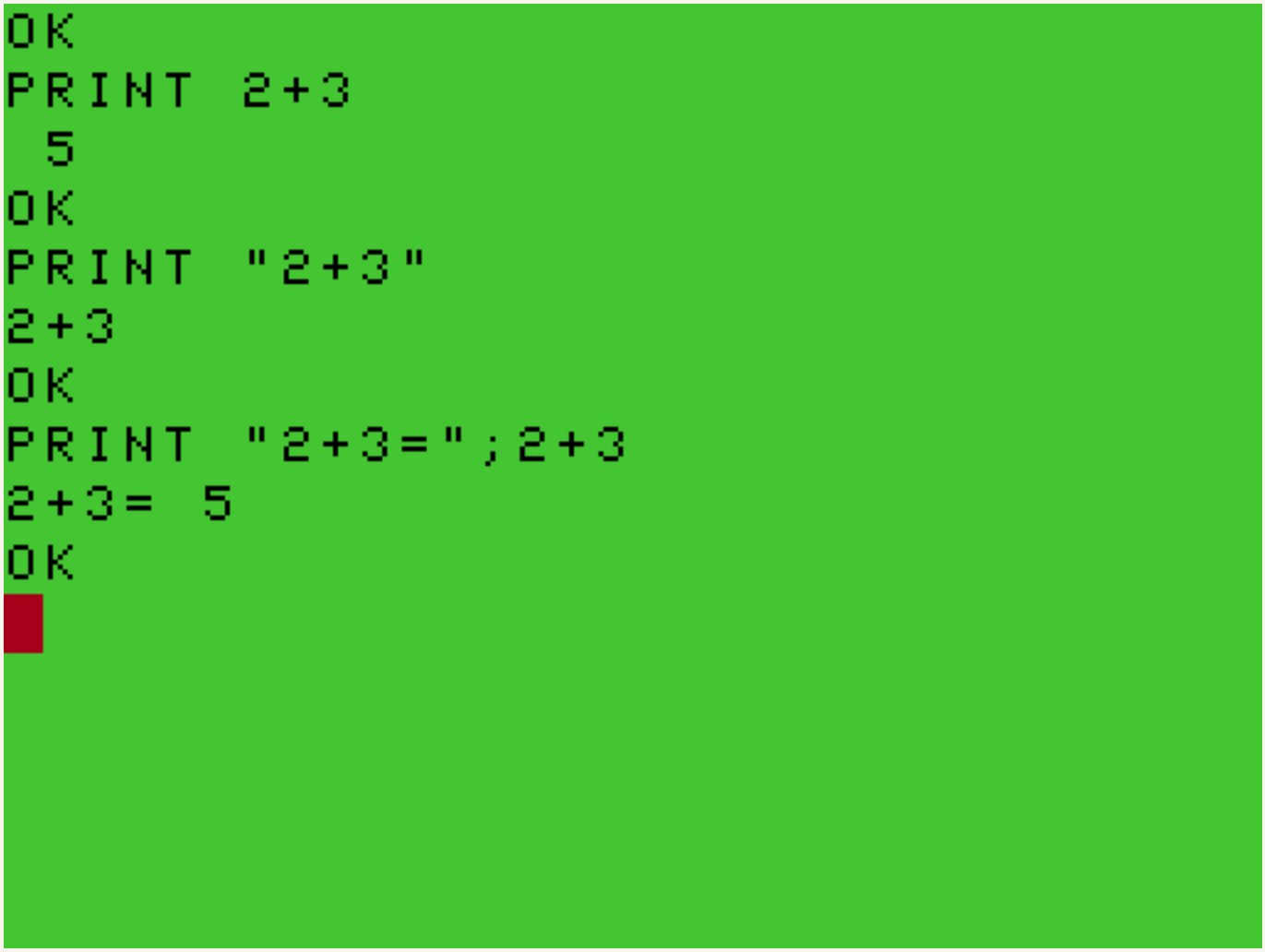


Figure 4: The CoCo shows three answers in order. First comes the number 5, then the text 2+3, then 2+3= 5 together on one line.

How did the CoCo know the word PRINT? Its **ROM**, short for *read-only memory*, holds a program called the **BASIC interpreter**. The interpreter recognizes BASIC words and symbols. A chip called the **CPU**, or *central processing unit*, carries out the interpreter's instructions. Your two plus signs gave it two different jobs on the very same line.

Try expressions at the BASIC prompt.

Where to go next

So far, pressing Return has meant "do this now." In chapter 3 you'll give the CoCo a message it can repeat on demand.

Curious about fitting more letters across the screen? Pick a planet uses a 40-column CoCo 3 display. You can come back to the Moon Base story whenever you like.

Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17, particularly PRINT.

Send a message from Moon Base

Chapter 3 of 7
Earth has come into range. Moon Base has a two-line message ready. Can you get the CoCo to hold both lines and send them to the screen on your signal?

Give each line a number

Continue at the CoCo prompt from chapter 2. The calculations you tried there have left you with an empty program. Click the screen or press **Option-Command-K**. Type these two lines, pressing **Return** at the end of each one.

```
10 PRINT "THIS IS MOON BASE."
20 PRINT "CAN YOU HEAR ME?"
```

Where's the reply? Look at the first thing you typed on each line. A **line number** tells BASIC to keep that instruction in RAM. Together the stored instructions make a **program**. Your words are ready in memory, waiting for the signal to start.

Send the signal

Type this command on a fresh line and press **Return**.

```
RUN
```

```
THIS IS MOON BASE.
CAN YOU HEAR ME?
OK
```

BASIC starts with the lowest line number. It carries out line 10, moves on to line 20, then hands the keyboard back to you at OK. The prompt is how you know this transmission is over.

Change the request

The crew has a more urgent question. Type a fresh line 20 exactly as shown and press **Return**.

```
20 PRINT "SEND SNACKS!"
```

How many instructions does the CoCo have now? Type this and press **Return** to see the stored program.

```
LIST
```

```
10 PRINT "THIS IS MOON BASE."
20 PRINT "SEND SNACKS!"
```

Count the line numbers. There are still two. BASIC used the number 20 to find the instruction you meant to replace. Line 10 was already in RAM, ready to use again.
Type **RUN** and press **Return**. Read the new transmission on your screen.

```
THIS IS MOON BASE.
SEND SNACKS!
OK
```

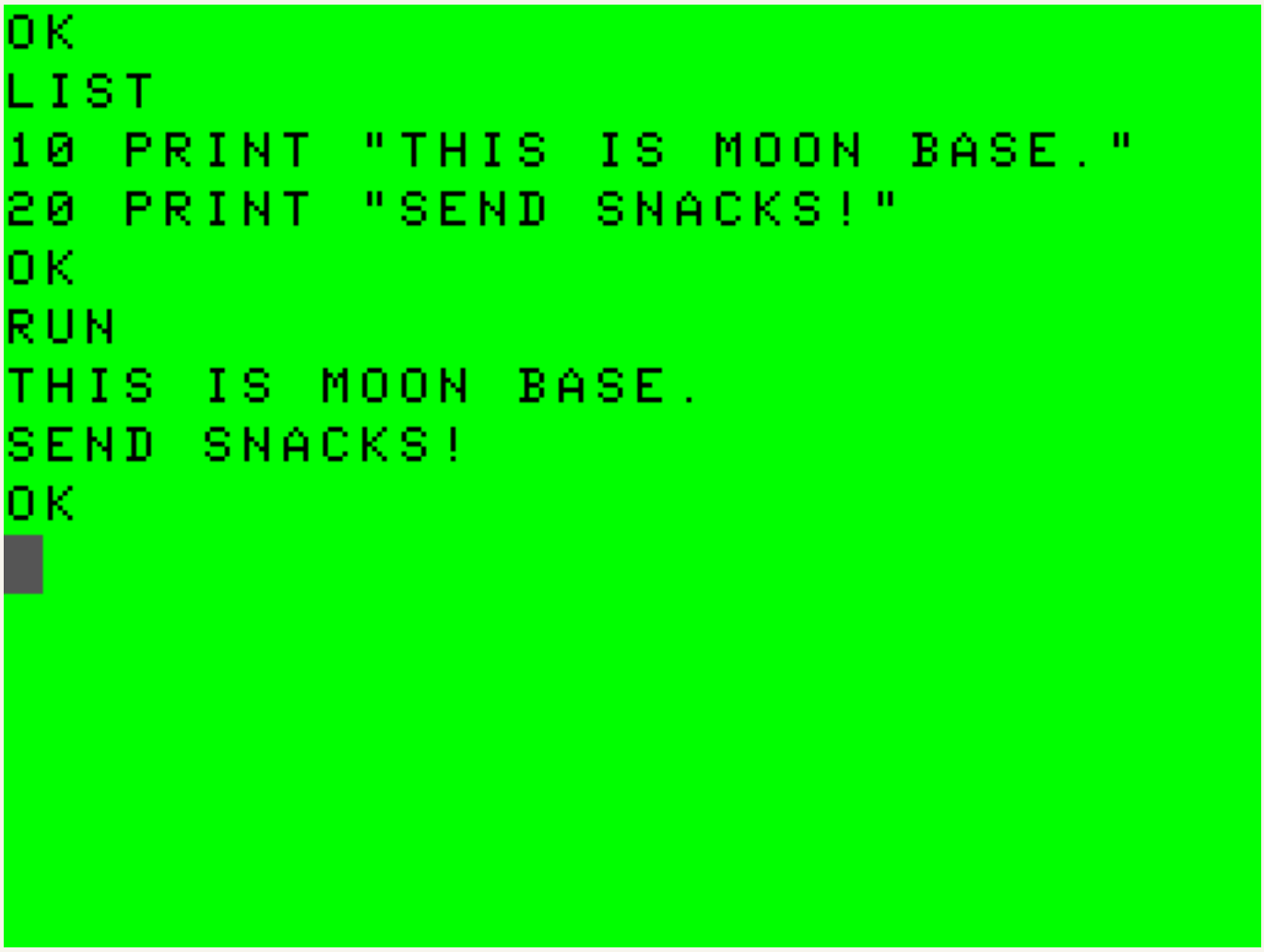


Figure 5: LIST shows the instructions with their line numbers. RUN sends the two messages, including the crew's new request for snacks.

If the old question still appears, use **LIST** again. Check the number and the words on the second line. You can retype that whole numbered line, press **Return**, and try **RUN** again.

Renumber a program and follow its branches.

Keep the beacon handy

At the moment your beacon lives in the CoCo's RAM. Chapter 4 will use that space for a different program. You can rebuild the beacon from the two lines shown under LIST, including the crew's snack order. In chapter 5 you'll give it a home on disk.

Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17, particularly LIST, PRINT, and RUN.

Pick a planet

Chapter 4 of 7

Two travelers run the same program. One gets a greeting from Earth and the other gets a greeting from Mars. Can you find the moment their trips take different paths?

Make two trips

Use the Disk BASIC session from chapter 3, with the starter disk in FD0, the first floppy drive. Loading the next program uses the RAM that held your beacon. Its two lines are in chapter 3 for when we need them again.

1. Click the CoCo display, or choose **Input → Focus CoCo Display** (Option-Command-K). At `OK`, type `LOAD"HELLO"` and press **Return**. Wait for `OK`.
2. Type `RUN` and press **Return**. The program asks `What planet do you live on?` and offers two choices.
3. Tap **3** by itself. The choices stay on screen. Now tap **1**. Read the greeting on the blue strip. What will change if you take the second trip?
4. At `OK`, type `RUN` and press **Return** again. This time tap **2**. Read the greeting on the red strip.

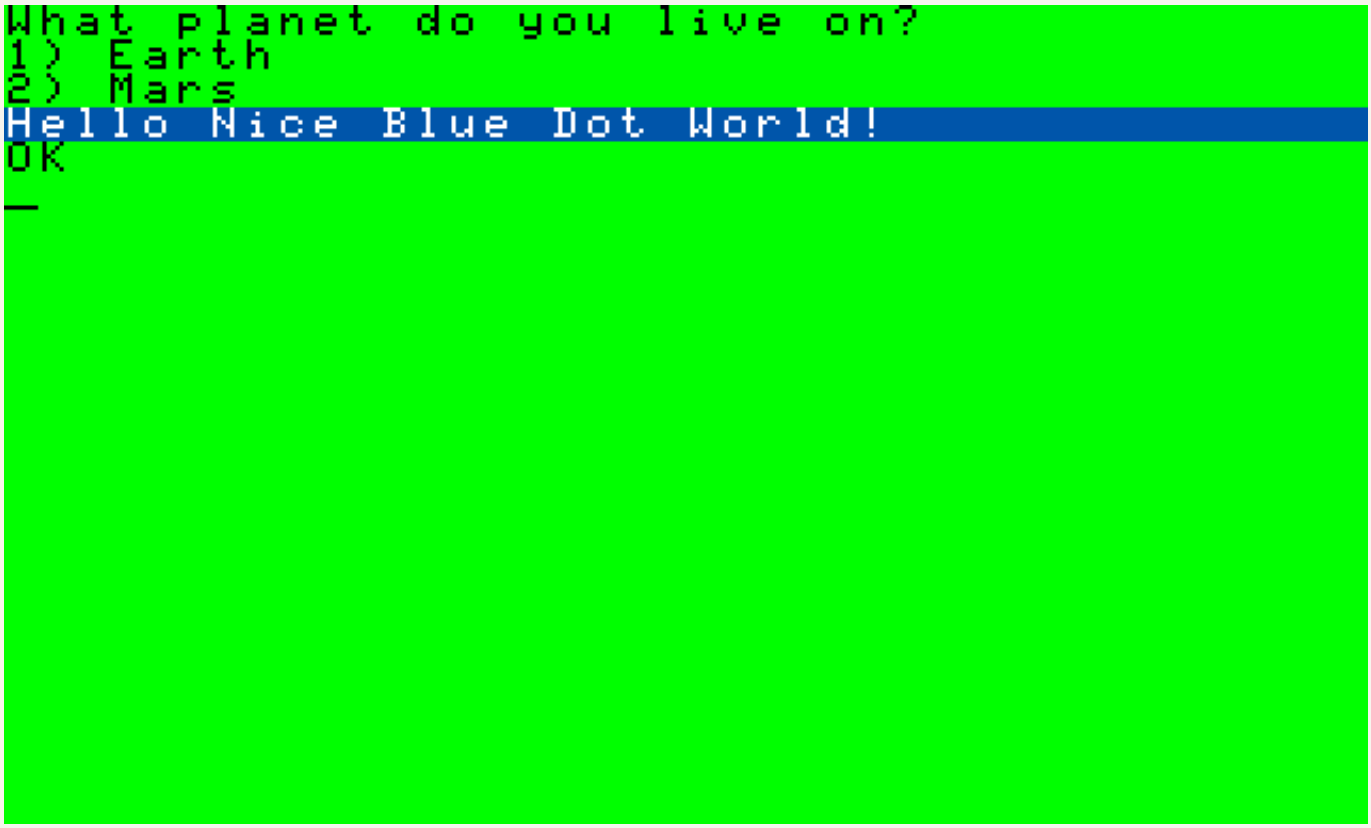


Figure 6: Choosing 1 for Earth prints “Hello Nice Blue Dot World!”



Figure 7: Choosing 2 for Mars prints “Hello Cold Red Dot World!”

At the prompt, **Return** tells BASIC that your command is complete. During the planet question, the running program reads one key at a time. It can act on **1** or **2** the instant you press it.

Follow the choice in the listing

A **listing** is the set of instructions you can read and edit. In VCC Plus choose **Develop** → **BASIC Development**. The CoCo is at `OK` with the planet program still in RAM, so choose **Read from CoCo**. If you opened the lesson separately, click the planet listing in the Manual Library to work on an editable copy. In a web browser, save the file and use **Open…** in BASIC Development. In the editor, choose **More file actions** (the three dots) → **Save As…** and give your working copy a fresh name such as `My Planet.bas`.

Find line 60. `INKEY$` checks the keyboard and `K$` holds the character it found. The dollar sign marks a name that holds text. Line 70 asks whether that text is `"1"`. If it is, BASIC jumps straight to line 110, the Earth greeting. If you pressed **3**, neither planet matches and line 80 sends BASIC back to line 60 to check again. That's what it was doing while the screen seemed still.

Look up at line 20. `WIDTH 40` gives the CoCo 3 room for 40 characters on a row, eight more than the 32 across the first screen you used. Each route chooses a color with `PALETTE` and puts a greeting in `M$`. Both routes meet at line 120. `ATTR 3,6` selects the colors for the letters and their background, and `PRINT M$` displays the chosen greeting.

Try 32, 40 and 80 columns.

Send a greeting of your own

1. Find line 110 in your Mac copy. Select only the words between its quotation marks and replace them with `Greetings from Earth!`. Your whole line should now read like this.

```
110 PALETTE 6,10:M$="Greetings from Earth!"
```

2. Choose the editor's **Save**, then its **Run**. If it asks about replacing the BASIC program in CoCo memory, check that you're working in the practice session and confirm.
3. Press **Option-Command-K** to return to the display. Tap **1** at the planet question. Look for `Greetings from Earth!` on the blue strip.

You changed the words while keeping the color choice. Where would you look for the words a visitor to Mars sees?

Find the Mars greeting

Line 90 pairs the Mars color with the words `Hello Cold Red Dot World!`. Both planets use the same `PRINT` on line 130.

For more on the display commands, see Tandy Corporation, *Tandy Color Computer 3 BASIC Quick Reference Guide* (1986), commands, pp. 2–17.

Send tomorrow a message

Chapter 5 of 7

Suppose someone comes to Moon Base tomorrow and wants to hear your beacon. Can you bring the message back after you empty BASIC's working memory?

Make a disk for Moon Base

A **disk image** is a Mac file that holds the contents of a whole CoCo disk. We'll copy the starter disk. It already has space prepared for BASIC files and a program called HELLO. Choose a fresh filename so you can recognize your disk tomorrow.

1. In your Disk BASIC practice session, choose **Media → Floppy Disks…**. Find **FD0** and check that it holds the starter disk from chapter 4. If you see a personal disk there, return to your practice setup before continuing.
2. In FD0 choose **Actions → Save As…**. Choose an easy-to-find folder and enter a fresh Mac filename such as `Moon Base.dsk`. If that name already exists, choose an unused one. Keep a note of the folder and name.
3. Read FD0 again. Look for **your new filename** and **Saves automatically**. If the new copy says **Temporary changes** or **Write protected**, open **Disk options** in its row and turn off that setting on your copy. Wait for any save to finish. Resolve any save error before continuing.
4. Return to the main window. If the CoCo is paused, choose its **Run**. Click the display or press **Option-Command-K** to type at the CoCo.

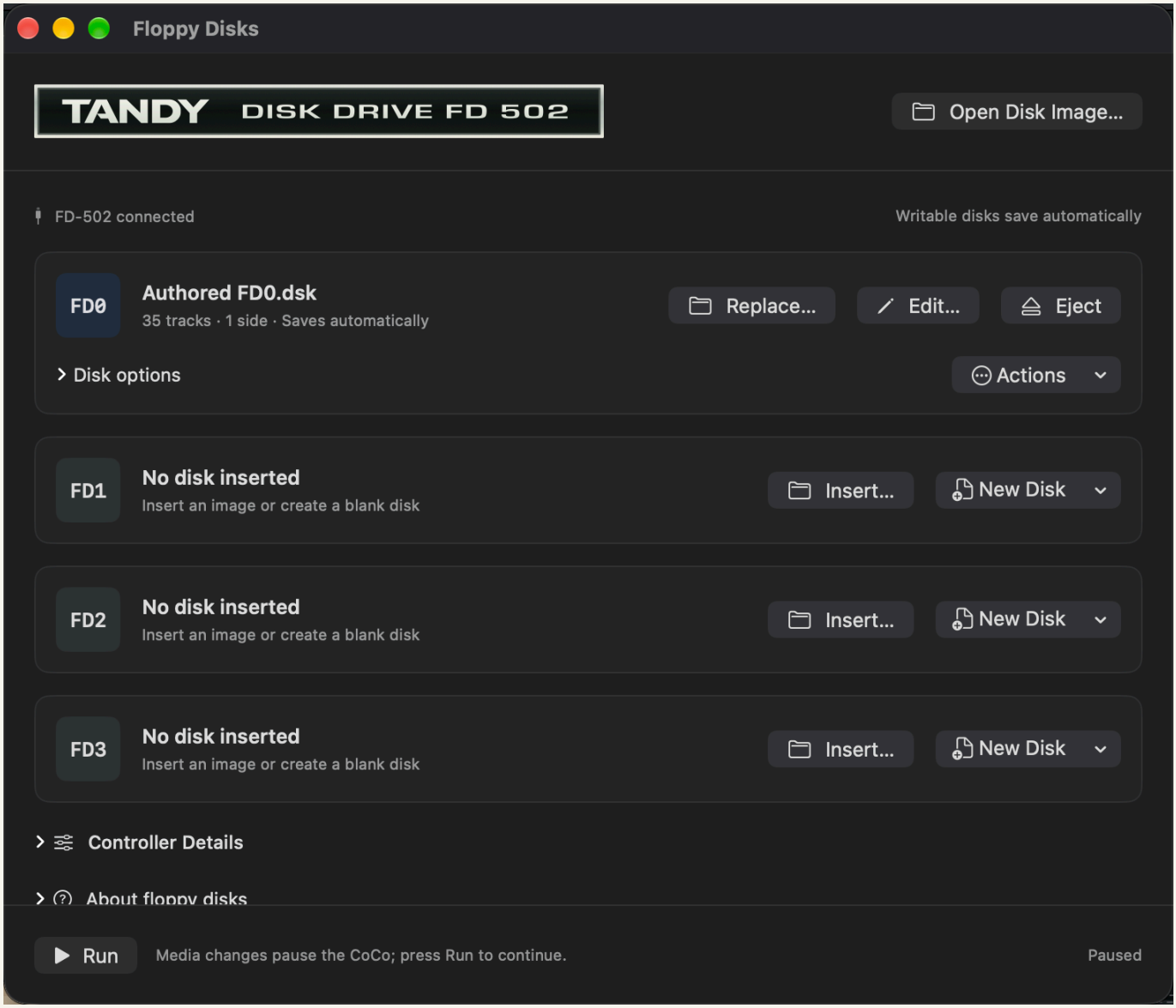


Figure 8: The current Floppy Disks window shows the filename and save mode in FD0. This example uses Authored FD0.dsk; your practice copy should show Moon Base.dsk (or the fresh name you chose) and Saves automatically.

To use these controls from the keyboard, **Input → Focus Mac Controls** (Option-Command-U) takes you out of the display. **Tab** and **Shift-Tab** move between Mac controls, and **Space** operates a focused button. If Tab skips buttons, turn on **Keyboard navigation** in **System Settings → Keyboard**.

Rebuild the beacon and save it

1. At `OK`, type `NEW` and press **Return**. This clears the planet program from BASIC's working memory.
2. Enter both numbered lines below, pressing **Return** after each one. Include the new snack request from chapter 3.

```
10 PRINT "THIS IS MOON BASE."
20 PRINT "SEND SNACKS!"
```

3. Type `RUN` and press **Return**. Check both lines of the message. If one needs a fix, retype its whole numbered line and run it again.

4. At `OK`, type `SAVE"BEACON"` and press **Return**. Wait for `OK`. This tells Disk BASIC to store the program on the disk in FD0.
5. Type `DIR` and press **Return**. This asks the disk for its directory, the list of files it holds. Find `BEACON` with `BAS` beside it. You should also see `HELLO`. Check the FD0 row for a completed save before clearing RAM.

If BASIC says `?WP ERROR`, the disk is write protected. Your two lines are still in RAM. Check the filename on FD0, open **Disk options** on your practice copy and turn off **Write protected**. If the switch is unavailable, use **Actions → Save As…** to make a fresh copy in your own folder. When the save finishes, use FD0's **Replace…** to open that new file. Turn off **Write protected** on the new copy if needed. When FD0 says **Saves automatically**, choose **Run** if paused and press **Option-Command-K**. Repeat the save and directory steps. For a Mac save error, resolve it using **Retry Save** or **Save As…** before going on.

Empty RAM and bring the message back

1. At `OK`, type `NEW` and press **Return**. Type `LIST` and press **Return**. The next `OK` appears with no numbered lines from that listing. Where could the two lines still be?
2. Type `LOAD"BEACON"` and press **Return**. Wait for `OK`. BASIC has copied the program from disk into RAM.
3. Type `RUN` and press **Return**. Check the reply.

```
THIS IS MOON BASE.
SEND SNACKS!
OK
```

Point to the three places involved. **RAM** holds the lines BASIC can run right now. **BEACON** is the name you look up with `DIR` on the CoCo disk. **Moon Base.dsk** is the file you'd look for in your Mac folder to find the whole disk again. Keep that disk in FD0 for your quiz. To see the same distinction with another program, find CARGO inside its disk image.

The editor's **Save** in chapter 4 wrote readable BASIC text to a Mac document. Here you typed `SAVE"BEACON"` at the CoCo prompt, so BEACON appears *inside* the disk image alongside HELLO.

For the disk commands and write-protection error, see *Color Computer Disk BASIC Quick Reference Guide* (InterTAN Canada Ltd., 1987), pp. 3–9.

Catch the computer in the act

Chapter 6 of 7

The beacon keeper wants three blinks. If the CoCo prints all three before you can look, how can you find out when it changed from blink one to blink two?

Give the signal a counter

This program prints a numbered message for each blink. Read from line 20 to line 40 with your finger. How many times do you think your finger will pass line 30?

```
10 CLS
20 FOR I=1 TO 3
30 PRINT "BLINK";I
40 NEXT I
50 PRINT "SIGNAL SENT!"
```

`I` is a **variable**, a name BASIC uses to keep a value in RAM. Here its job is to count. `FOR I=1 TO 3` starts the count at 1. `PRINT "BLINK";I` puts a word and the current number on the screen. `NEXT I` adds one to the counter and sends BASIC around again while there is another turn left. A group of instructions that repeats this way is a **loop**.

In **Help → Manual Library...**, click SIGNAL.bas. An editable copy opens in a **BASIC Example** window. Keep the book beside it so you can read and step through the program together. The CoCo keeps doing what it was doing until you choose **Run**.

Reading in a web browser? Save a copy of the linked file on your Mac, then choose **Develop → BASIC Development → Open...** in VCC Plus to open it. Save any previous work if the editor asks.

Count by twos or count backward.

Stop before the first blink

1. In the editor, click the circle in the left margin beside line **30**. You've set a **breakpoint**, a request for the CoCo to pause when it reaches that line. Using the keyboard, move the insertion point to line 30 and activate the toolbar button **Toggle BASIC breakpoint at insertion point**. Use either method once.
2. Expand **Variables and Watches** below the listing if it is closed. Choose **Watches**, type the capital letter `I` in its field and choose **Add** or press **Return**. A **watch** lets you read a value at each stop.
3. Choose the editor's **Run**. Choose **Replace and Run** if asked about replacing your previous practice program in RAM. Look for the highlight at line 30 and the value **1** beside `I`. Line 10 has cleared the CoCo screen. The first BLINK is still to come.

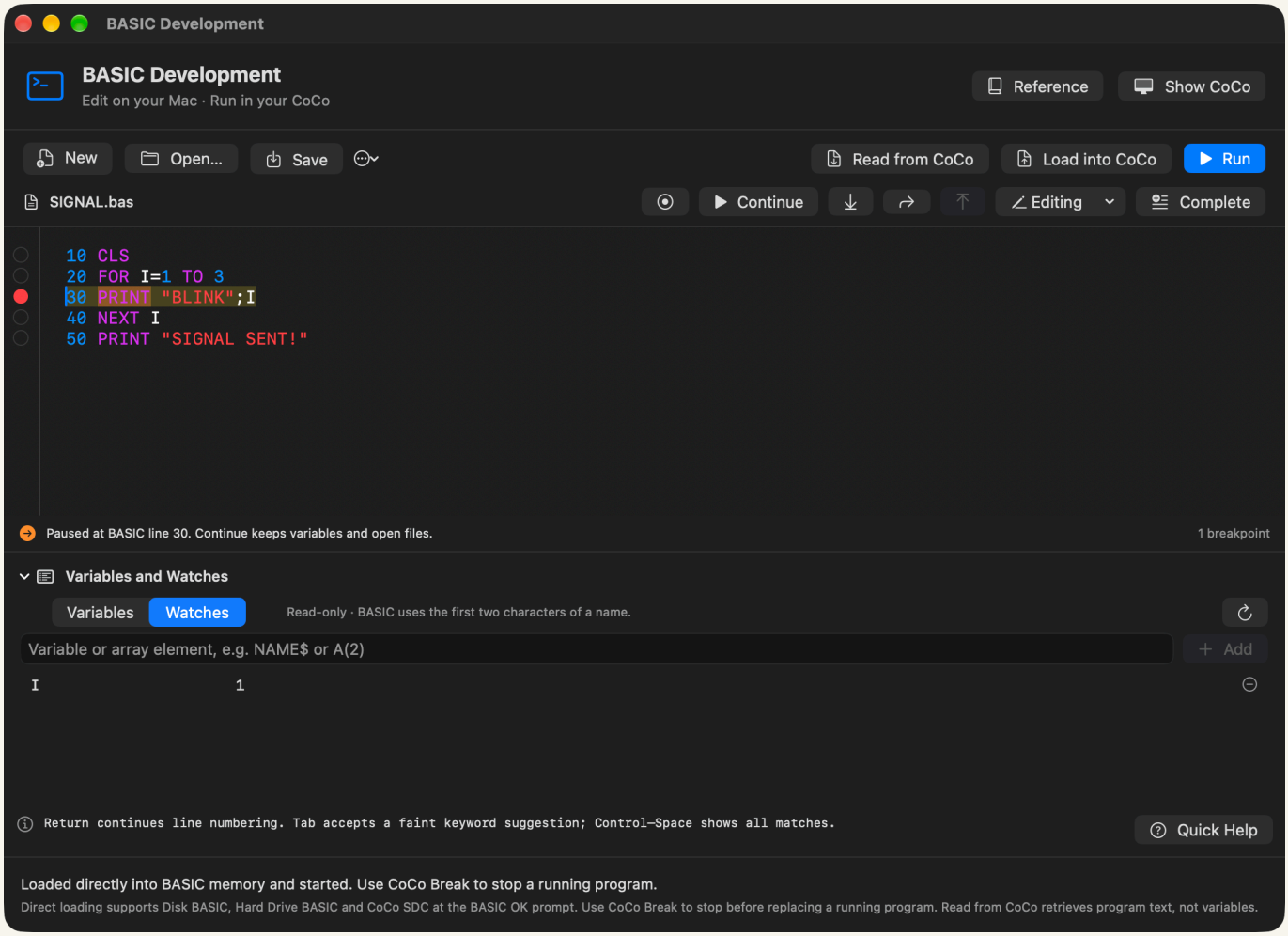


Figure 9: In BASIC Development the line highlight shows where execution paused. The watch gives the value of I at that stop.

Choose **Step Into BASIC statement**, the down-arrow button beside **Continue**, once. The highlight moves to line 40. On the CoCo display, `BLINK 1` has appeared. Read the watch again. It still says **1**. Which instruction will change it?

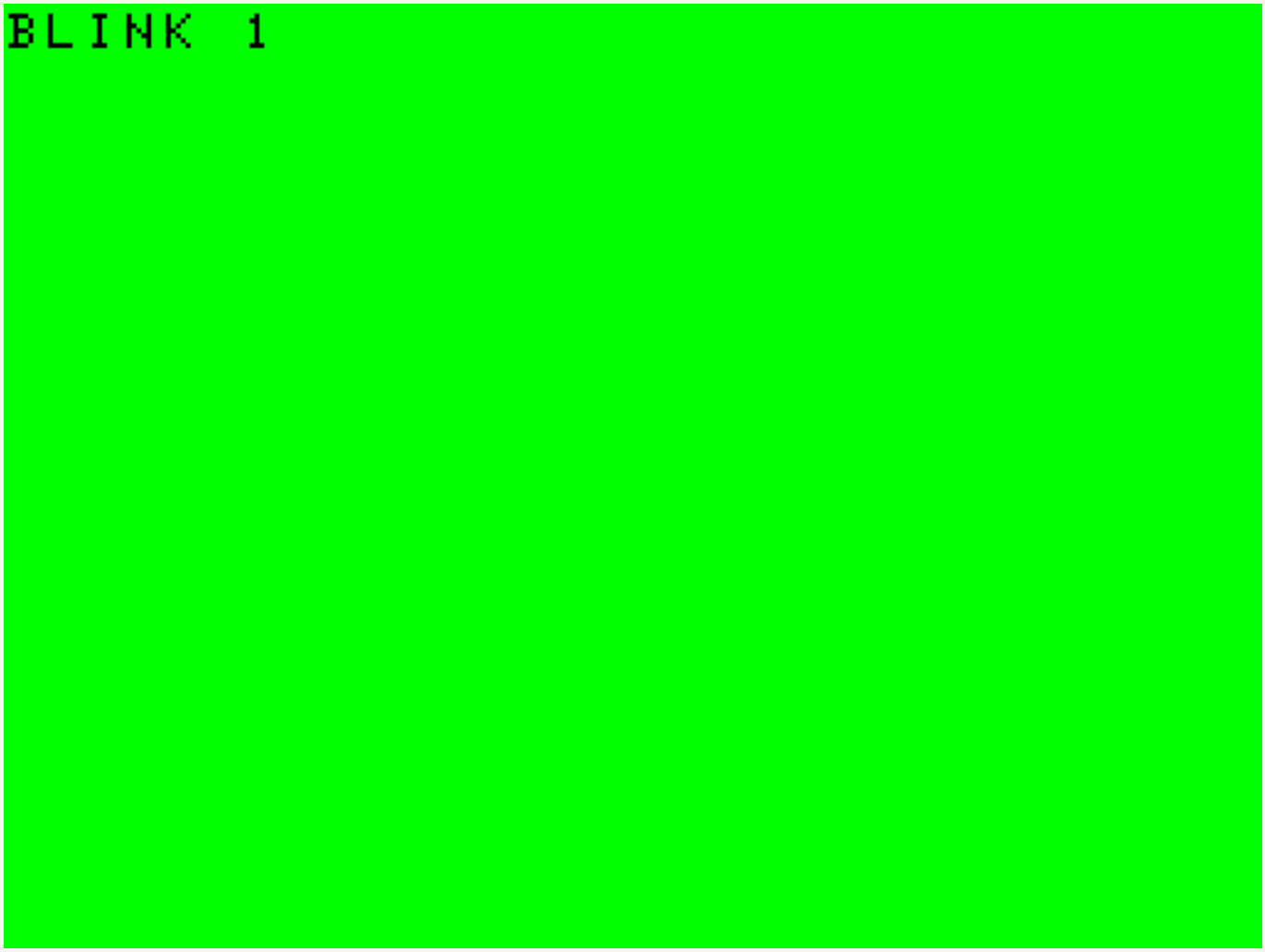


Figure 10: After one step the CoCo shows BLINK 1. BASIC has printed the first message and is waiting before NEXT I.

If all three blinks print in one burst, your program has finished. First check that the marker is beside line 30 and you used **Run** in BASIC Development. If it happens again, save any work you want to keep. In your practice session, choose **Cold Restart – Clear RAM…**, confirm, and wait for BASIC's **OK**. Open SIGNAL in the editor, check the line 30 breakpoint and run it from there.

To see the guest screen if the editor covers it, choose **Show CoCo**. Use the **Window** menu to bring the editor with SIGNAL back to the front. Choose **Help → Manual Library…** to bring this lesson back. The Mac controls work while the emulated CoCo waits.

Predict the next two stops

Choose **Continue** once. This resumes your paused program. Before looking at the screen, predict what has changed. Line 30 is highlighted again. The watch says **2**, but only `BLINK 1` has printed. The highlighted line is the instruction BASIC is *about to* carry out.

Choose **Continue** once more. Stop and count the messages before you peek at the watch. Now you should have two messages and a counter of **3**. What will appear if you continue one last time? Make your prediction, then choose **Continue** a third time.

Check the finished signal

The CoCo prints the last blink, finishes the loop and carries on to line 50. Here is the complete screen text.

```
BLINK 1
BLINK 2
BLINK 3
SIGNAL SENT!
OK
```

When you see `SIGNAL SENT!` followed by `OK`, the CoCo is ready for the quiz.
Watch a subtraction one statement at a time.

How a number gets onto the screen

At the second stop on line 30, RAM held the number 2 under the name `I`. The screen still showed the earlier number 1. When BASIC carried out `PRINT`, it turned the current number into a character you could read. You saw those two jobs happen at different times.

Memory stores patterns of **bits**. A bit can be 0 or 1, and eight bits make one **byte**. The CoCo uses patterns in memory to represent the characters on its screen. BASIC also uses space in RAM to keep a variable's name, the kind of value it holds, and the value itself. A one-letter name on your watch is enough to look up all that information.

To watch a number change one instruction at a time, follow a byte through register A. To see how RAM becomes a picture, look for a dot on a hidden page.

For the loop commands, see Tandy Corporation, *Tandy Color Computer 3 BASIC Quick Reference Guide* (1986), commands, pp. 2–17.

Give your friend the controls

Chapter 7 of 7

You get to make the entrance quiz for the Moon Base crew. A friend gets the keyboard. Can the CoCo ask the questions, keep the score, and play your game again after a restart?

Make the first point count

Start in your Disk BASIC practice session with the Moon Base disk from chapter 5 in FD0. Finish the signal program so you can see `OK`. Click the CoCo display or press **Option-Command-K**. This time you'll build the game at the CoCo keyboard.

1. Type `NEW` and press **Return** to make room in RAM for the quiz.
2. Type these seven lines, pressing **Return** after each one. Use the digit zero in `S=0`. Use straight double quotation marks around each piece of text.

```
10 CLS
20 S=0
30 PRINT "MOON BASE CREW QUIZ"
40 PRINT "WHICH PART SHOWS A MESSAGE?"
50 PRINT "1 SCREEN   2 KEYBOARD"
60 INPUT A
70 IF A=1 THEN S=S+1
```

3. Type `RUN` and press **Return**. At the question mark, type **1** for SCREEN and press **Return**. This short version ends here and gives you `OK`.
4. At `OK`, type `PRINT S` and press **Return**. Look for **1**. The program kept a point even though you hadn't yet written the line to display the score.

Line 60, `INPUT A`, waits for a number and **Return**, then stores the number under the name `A`. Line 70, `IF A=1 THEN S=S+1`, makes a decision. Read its first part as a question. “Is the answer in A equal to 1?” If it is, BASIC does the part after `THEN`. It takes the old score in `S`, adds one and stores the result back in `S`. A different answer skips that addition. The first equals sign **compares**. The second **stores**.

Follow an answer through a two-planet menu.

Add two more questions

Keep the first seven lines in RAM. At `OK`, enter the remaining lines below, one at a time. The higher line numbers put them after line 70. Line 110 awards a point for answer **2** because DISK is the second choice on that question.

```
80 PRINT "WHICH KEEPS A SAVED PROGRAM?"
90 PRINT "1 RAM   2 DISK"
100 INPUT A
110 IF A=2 THEN S=S+1
120 PRINT "WHAT DOES PRINT 2+3 SHOW?"
130 PRINT "1 FIVE   2 THE TEXT 2+3"
140 INPUT A
150 IF A=1 THEN S=S+1
160 PRINT "YOUR SCORE";S;"OUT OF 3"
170 END
```

Run the game twice. Each time, type `RUN` at `OK` and press **Return**. Give *one number and Return per question*. Try **1, 2, 1** on the first run. Try **2, 1, 2** on the second. Predict each final score.

Check the two scores

On the first run, look for `YOUR SCORE 3 OUT OF 3`. On the second, look for `YOUR SCORE 0 OUT OF 3`. Line 20 sets the score to zero at the start of every game. Each `INPUT A` replaces the previous answer, while `S` keeps the points earned so far. Line 160 prints the score.

If a score surprises you, type `LIST` at `OK`. Compare lines 70, 110 and 150 with the listing. Retype a whole numbered line to correct it, then run both tests again.

Open the complete runnable quiz

QUIZ.bas contains the entire original program. Click its name in the Manual Library to read the complete listing in its own editable window. In a web browser, save a copy and use **Open...** in BASIC Development.

```
10 CLS
20 S=0
30 PRINT "MOON BASE CREW QUIZ"
40 PRINT "WHICH PART SHOWS A MESSAGE?"
50 PRINT "1 SCREEN   2 KEYBOARD"
60 INPUT A
70 IF A=1 THEN S=S+1
80 PRINT "WHICH KEEPS A SAVED PROGRAM?"
90 PRINT "1 RAM   2 DISK"
100 INPUT A
110 IF A=2 THEN S=S+1
120 PRINT "WHAT DOES PRINT 2+3 SHOW?"
130 PRINT "1 FIVE   2 THE TEXT 2+3"
140 INPUT A
150 IF A=1 THEN S=S+1
160 PRINT "YOUR SCORE";S;"OUT OF 3"
170 END
```

Give the game a title and a home

1. At `OK`, type this replacement line and press **Return**.

   ```
   30 PRINT "SPACE CADET QUIZ"
   ```

2. Type `RUN` and press **Return**. Look for `SPACE CADET QUIZ`, then answer all three questions to return to `OK`.
3. In **Media → Floppy Disks...**, check the filename in FD0 against the practice disk you made. Check for **Saves automatically**. Press **Option-Command-K** to focus the CoCo, then type `DIR` and press **Return**. On your chapter 5 disk you'll see HELLO and BEACON. If there's already a QUIZ you want to keep, make a fresh disk copy with FD0's **Actions → Save As...** before saving this version. If you made a copy, choose **Run** if paused, then press **Option-Command-K**.
4. At `OK`, type `SAVE"QUIZ"` and press **Return**. Wait for `OK`. Type `DIR`, press **Return**, and find the entry with `QUIZ` and `BAS`.
5. Check FD0 once more and wait for the disk save to finish. For a write-protection or save error, keep your game in RAM and follow the disk recovery steps before you restart.

Let someone play after a restart

1. In the main VCC Plus window open **Restart → Cold Restart – Clear RAM...**. Now that your game is saved, confirm **Cold Restart**. This restarts the CoCo and clears its working memory.
2. Check that your practice disk is still in FD0. If the machine is paused, choose the main window's **Run**. Return to the CoCo display and wait for `OK`.

3. Type `LOAD"QUIZ"` and press **Return**. Wait for `OK`, then type `RUN` and press **Return**. Your new title should appear.
4. Move the book and listing out of sight. Give your friend the keyboard. Tell them, “Choose 1 or 2 and press Return for each answer.” Let the CoCo tell them their score.

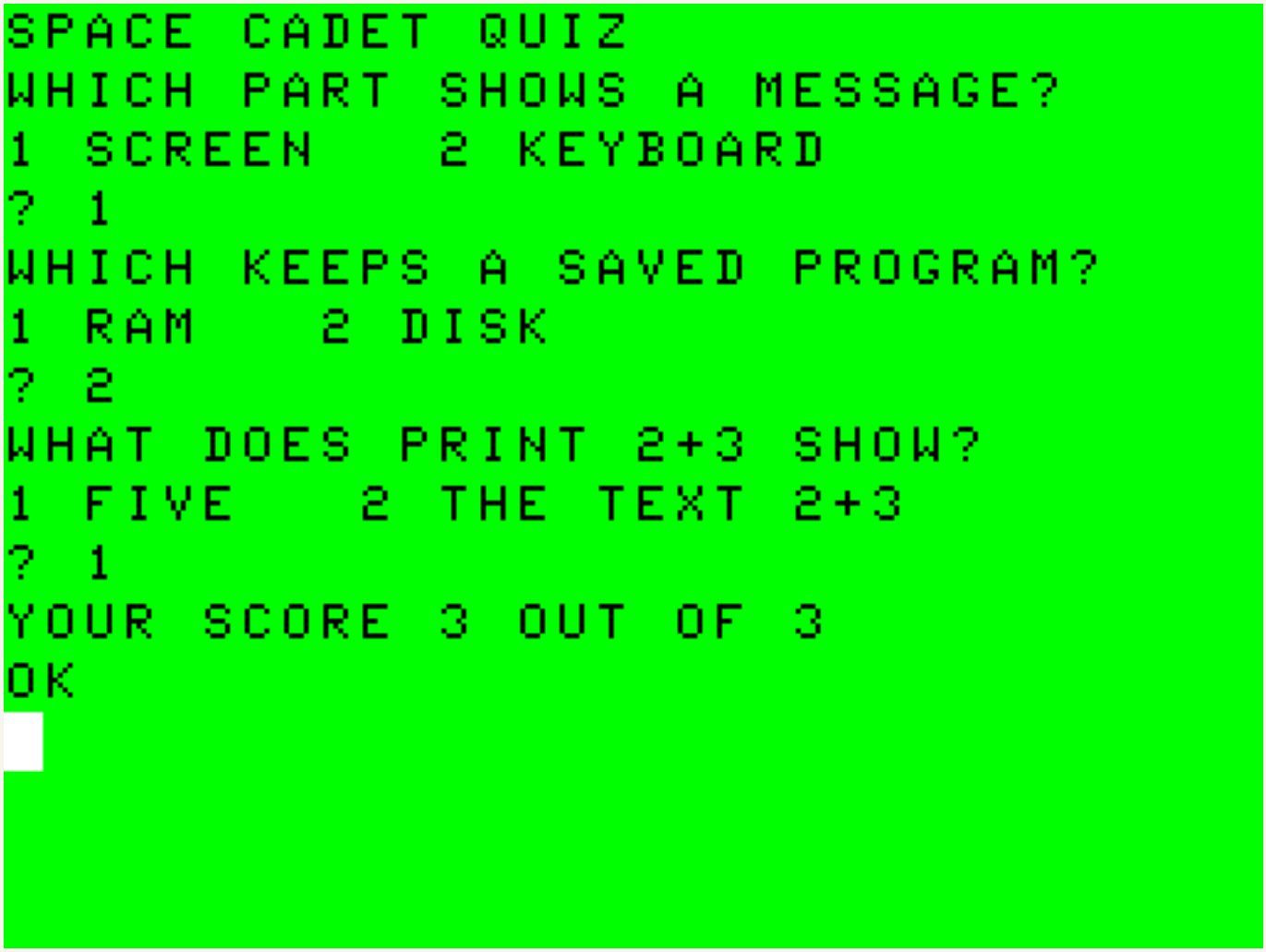


Figure 11: The customized game runs from disk with the title SPACE CADET QUIZ and prints a final score.

Ask your friend to pick one answer they gave. Can you tell its story? The keyboard supplied a number, `INPUT` put it in RAM, and the CPU followed your `IF` instruction to decide whether to add a point. The screen showed the total. And when the restart cleared RAM, your disk held the program, including the title you wrote.

If you want to write a question about your friend, keep this working game on the Moon Base disk. Make another practice copy, change one question and its choices, and decide which answer earns the point. Play it once each way to see whether your rule works.

For the language commands, see Tandy Corporation, *Tandy Color Computer 3 BASIC Quick Reference Guide* (1986), pp. 2–17. For saving and loading, see *Color Computer Disk BASIC Quick Reference Guide* (InterTAN Canada Ltd., 1987), pp. 3–8.

Your field guide

Appendix A
Come here if you remember what you wanted to do but have forgotten what BASIC calls it.

Find a command

When you want to…	Use	Try it in
Put words or a number on screen	`PRINT`	Give one instruction
See the program in memory	`LIST`	Moon Base
Start the program again	`RUN`	Moon Base
Wait for one key	`INKEY$` inside the program	Pick a planet
Ask for a number and wait for Return	`INPUT` inside the program	The crew quiz
Make a choice	`IF` and `THEN`	The crew quiz
Count trips through a loop	`FOR` and `NEXT`	Catch the computer
Find files on the CoCo disk	`DIR`	Tomorrow's message
Keep the program or bring it back	`SAVE` or `LOAD`	Tomorrow's message
Clear the program in RAM	`NEW`, **after saving**	Tomorrow's message

When the screen surprises you

Typing goes nowhere. Look at the main emulator window. If the toolbar offers **Run**, the CoCo is paused. Resume it, then choose **Input → Focus CoCo Display**. If the BASIC editor is stopped at a breakpoint, use **Continue** there to keep your place in the program.

The planet menu waits; the quiz shows a question mark. The planet program watches for one key. Press 1 or 2. The quiz uses INPUT. Type a number and press Return so BASIC knows your answer is finished.

BASIC reports a syntax error. Compare the spelling, the quotes, and the number at the start of the line with the listing. Retype the whole numbered line. Run the program again. At the prompt, an instruction without a number takes effect as soon as you press Return.

The disk seems empty. Check that the selected profile is Disk BASIC and that FD0 shows your *practice* disk's Mac filename. Use DIR to see the names inside it. A new disk image and a copy of the initialized starter disk are different things. These lessons use a copy. Go back to the safe copy steps before writing.

The screen says ?WP ERROR. Stop before clearing RAM. Check the practice disk's name and write protection in Floppy Disks. The program you tried to save is still in RAM. Fix the destination and try the save again.

Files you can open

Reading in **Help → Manual Library…**? Click a filename to open an editable BASIC example beside the book. If you're reading in a browser, save the file to your Mac and choose **Open…** in **Develop → BASIC Development**. BASIC's LOAD command on the green screen looks *inside the mounted CoCo disk*.

- BEACON.bas – the first two lines from Moon Base.
- planet.bas – the two-color Earth or Mars starter.
- SIGNAL.bas – the three-blink counter.
- QUIZ.bas – the completed crew quiz with its original title.

Words from the workbench

Input is information going in, like the key you pressed. **Output** is information coming out, like the greeting you saw. A **prompt** is a sign the computer is ready for you.

An **instruction** tells a computer to do something. A **program** gives it a set of instructions to follow. The **CPU**, or central processing unit, carries out machine instructions. The **interpreter** is the program that understands your BASIC and directs the work.

RAM is read/write working memory. Your beacon lived there while you edited it. **ROM** is read-only memory. It holds built-in software, including the interpreter that understands PRINT. A **bit** holds one of two values. Eight bits make a **byte**.

A **file** is a named collection of data you can come back to. A CoCo **disk drive** reads and writes files on a disk. VCC Plus puts the contents of a whole disk in a **disk image**, one file on the Mac.

A **variable** has a name and holds a value. A **loop** repeats instructions. A **breakpoint** tells the debugger to pause before a chosen line runs. The debugger's **watch** shows a variable's value while the program is stopped.

On the bookshelf

Appendix B

The CoCo in this book has two histories. One belongs to the Tandy computer people took home in the 1980s. The other belongs to the people who made it possible to sit down at that keyboard again on a Mac.

If you want to go further

For more projects with color, loops and choices, open Programming in CoCo BASIC on the Color Computer 3. If you are curious what the processor did between two blinks, follow a byte inside the 6809. For disks, tapes and cartridges, carry a program between devices. See the manual library.

Already inside VCC Plus? **Develop → BASIC Quick Reference** gives you command syntax and small examples. You can keep your own program open while you look something up.

Bibliography

- CoCopedia contributors. “Color Computer 3.” *CoCopedia*, revision 11134. Community history of the machine and its hardware.
- Tandy Corporation. *Introducing Your Color Computer 3*. 1986. “To Our Customers” and the introduction to the hardware.
- Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17; functions, pp. 18–22.
- *Color Computer Disk BASIC Quick Reference Guide*. InterTAN Canada Ltd., 1987. Disk commands, pp. 3–8; errors, p. 9.

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

About the pictures

The hardware drawing uses VCC Plus’s original artwork. The application credits explain its origins and permissions. Artwork credits.

The window pictures show VCC Plus on a Mac. A CoCo screen capture shows just what the guest computer draws. Sizes vary with your Mac window and display settings. The commands and results are also given as text so you can read them at your preferred size.

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

Back at the CoCo

Put a disk in drive zero. Find your way around a Multi-Pak. See a familiar picture change when you switch monitors. Your old CoCo habits have a place on this Mac.



Cream Tandy FD-502 enclosure with two black floppy drives

Brian K. Holman

Book 2 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Which CoCo did you bring with you? 3
Bring up a disk you remember 5
Tape, cartridges and the slot switch 9
Which drive zero? 12
Same bytes, different monitor 16
Visit a disk and come home 20
Beside the keyboard 22
Behind the desk 24
Publication and artwork notices 25

Which CoCo did you bring with you?

Chapter 1 of 6

Maybe your fingers remember exactly where BREAK was. Maybe you remember pressing Play on a cassette recorder, then waiting for a name to appear on the TV. Bring those memories to the keyboard. You can use them to find your way around this CoCo, even if the one you owned had a different number on its case.

Find your old hardware on the Mac

Choose **Display → Interactive System** in VCC Plus. The computer sits in the keyboard case beneath the monitor. Read the badge above the scene for its RAM and the badge on the right for its processor. The **Disk BASIC** starter setup pairs a 6809 with 512 KB of RAM. Follow the disk drive back to the cartridge beside the keyboard. That cartridge is the floppy controller. It gives BASIC a way to talk to the drive.

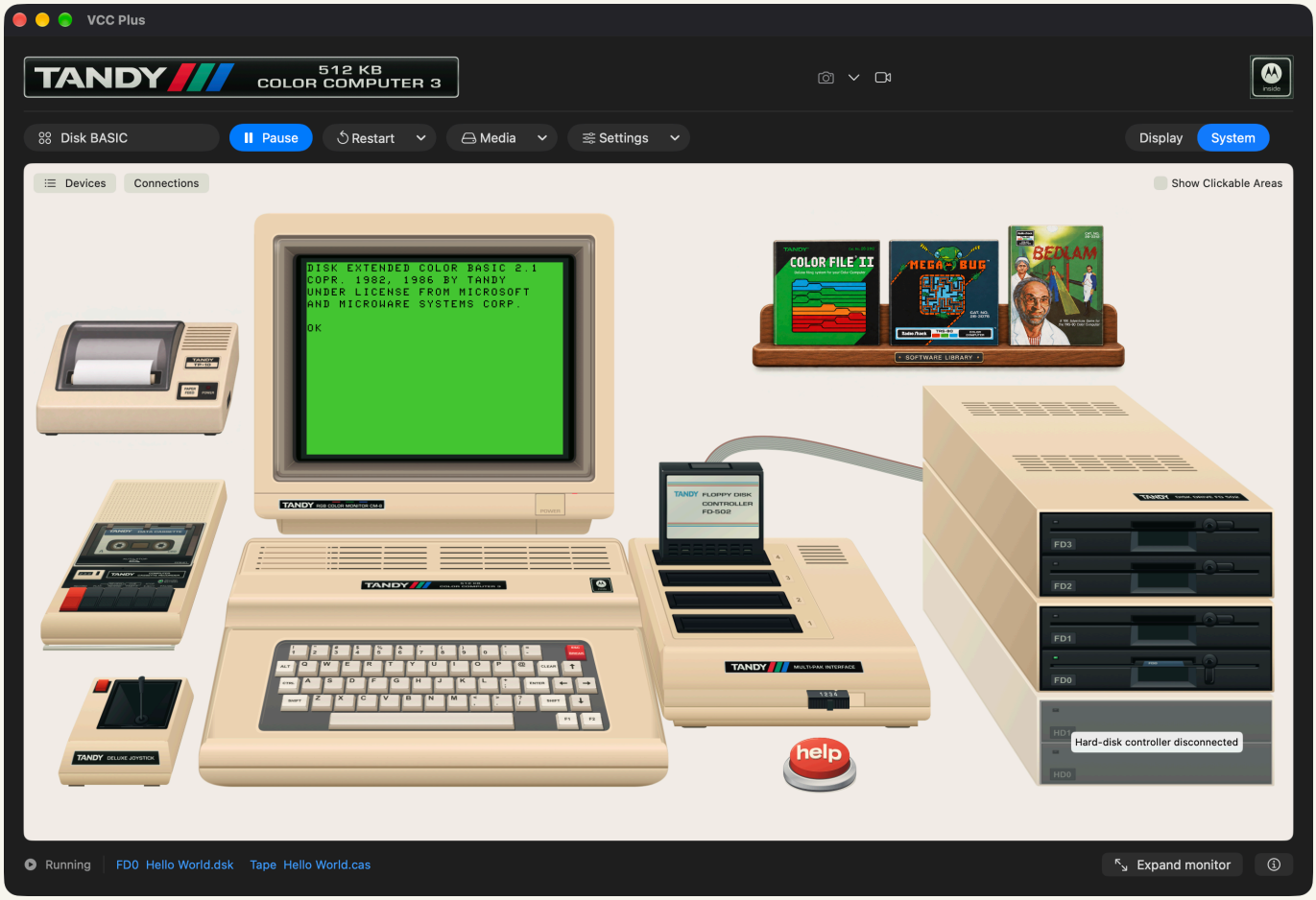


Figure 1: The System view places the monitor over the CoCo 3 keyboard, with the recorder on the left and the disk controller and drives on the right.

If you remember a CoCo 1 or 2, the green 32-column BASIC screen should feel familiar. Watch what happens in the next chapter when a program asks the CoCo 3 for 40 columns. If you came from a CoCo 3 with an RGB monitor, you can also try the same program on two video signals. For a closer look at the original keyboard case and its 1986 hardware, visit Find the keyboard and monitor.

What you used to do	Where to do it here	How you can tell
Insert a disk in drive 0	**Media** → **Floppy Disks...**, then **Insert...** or **Replace...** in FD0.	The Mac filename appears on the FD0 row.
Cue a tape	**Media** → **Cassette...**, then **Rewind** and **Play**.	The counter returns to the beginning. BASIC can start the motor.
Power up with a cartridge	**Cartridge** → **Cartridges...**. Choose the cartridge and its startup slot.	After the restart, a bootable cartridge displays its own opening screen.
Press BREAK	Focus the CoCo display and press **Esc**.	Running BASIC can report the line where it stopped.

Give BREAK its first job

Start in a **Disk BASIC** session with an empty program. If another program is in RAM, save it first. To load the starter setup, open **Machine** → **Configuration Profiles...**, select **Disk BASIC**, and choose **Load...**, then **Load Profile**. Press the Mac window's **Run** if the CoCo is paused.

Click the CoCo screen or press **Option-Command-K** (**Input** → **Focus CoCo Display**). At `OK`, type these two lines, pressing Return after each.

```
10 GOTO 10
RUN
```

The CoCo is busy doing exactly what you asked. Every trip to line 10 sends it back to line 10. Click **Pause** on the Mac toolbar. The whole machine freezes. Click **Run** alongside the toolbar controls and the CoCo picks up the loop again. Can you get BASIC's attention without restarting the machine?

Focus the CoCo screen and press **Esc** once. BASIC steps out of the loop and answers:

```
BREAK IN 10
OK
```

Type `LIST` and press Return. The loop is still there. BREAK gave you back the prompt. **Pause** let you take a breather in the middle of a running machine. That's also why the Mac's **Run** button and typing `RUN` at the BASIC prompt do different jobs.

To leave the display using only the keyboard, press **Option-Command-U** (**Input** → **Focus Mac Controls**). Tab and Shift-Tab move between controls. Space operates a focused button. If Tab skips buttons, turn on **Keyboard navigation** under **System Settings** → **Keyboard** on the Mac.

Choose your way back

- **I have a disk I want to run.** Begin with Bring up a disk you remember. Try our disk first, then make a safe visit to your own.
- **I want my old setup back.** Start with Tape, cartridges and the slot switch. Follow it with Which drive zero? to pick your system.
- **I used to write programs.** Try the same program from a disk and from the Mac editor. Then use your CoCo to compare composite and RGB.

Bring up a disk you remember

Chapter 2 of 6

On the label outside the disk it says **Port Zero**. In the CoCo's directory it says **PORT**. Which name do you type after LOAD? By the end of this trip you'll have greeted three different visitors with the same five-line program, and the disk will still be just as you found it.

Put the practice disk in FDo

Start in **Disk BASIC**. Save any program in RAM you want to keep, since loading PORT will replace it. Save a working copy of Port Zero.dsk in a folder you can find again in Finder. For a disk from your own collection, duplicate the image in Finder first and put the original aside.

1. Open **Media → Floppy Disks…**. Find **FD0**, the first drive. Choose **Replace…** if a disk is already there, or **Insert…** if the drive is empty. Choose your working copy in the Mac file picker. If a pending change needs attention, save it before replacing the disk.
2. Read the filename beside FD0. Open **Disk options** on that row and turn on **Write protected**. This lets the CoCo read the disk and makes the drive refuse writes. Leave the separate **Temporary changes** setting alone for this exercise.
3. If the footer reads **Paused**, click **Run** there. Return to the emulator and press **Option-Command-K** to type at the CoCo.

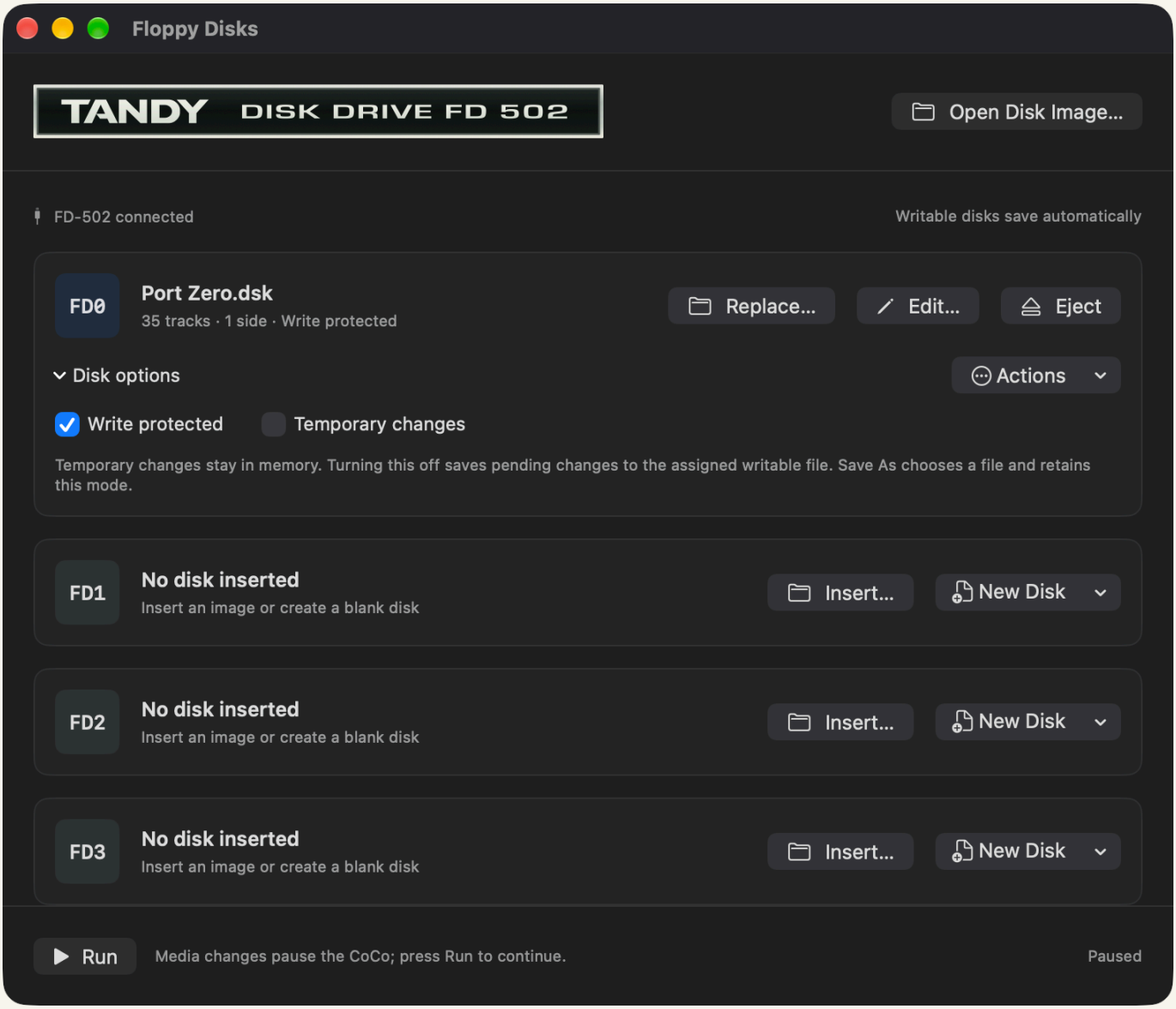


Figure 2: FD0 holds Port Zero.dsk. Its status says Write protected, the matching box is checked, and Run is available at the bottom because the CoCo is paused.

Type `DIR` at `OK` and press Return. Find **PORT** with **BAS** beside it. The drive holds a whole disk in one Mac file. BASIC just showed you a program *inside* that file.

Two ways to run PORT

First, use the floppy. Type `LOAD"PORT"` and press Return. When BASIC comes back with `OK`, type `RUN` and press Return. The display switches to 40 columns. Port Zero asks for your call sign. Type **ACE** and press Return. Look for this reply:

```
GOOD TO HAVE YOU ABOARD, ACE!
```

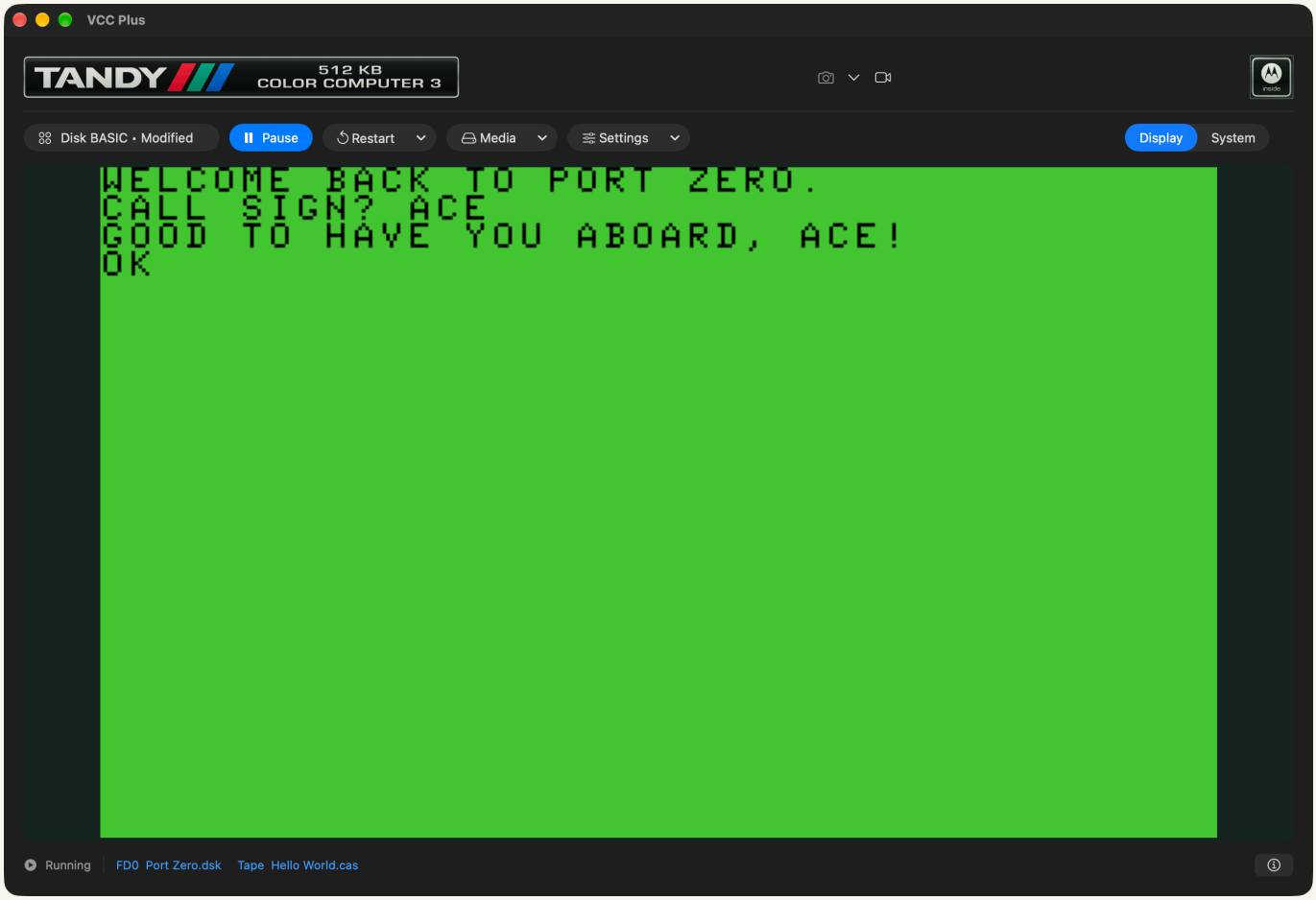


Figure 3: The live CoCo in Display view greets ACE and returns to OK. The status bar still names Port Zero.dsk in FD0.

Try a write on **this protected practice disk**. At `OK`, type `SAVE"LOG"`. The expected reply is `?WP ERROR`. That is the drive doing its job. The program you loaded is still in RAM. Type `RUN` again and use **SKY** for your call sign. The greeting changes to SKY. You have run the program twice and kept the original disk intact.

Now take the Mac route. Leave the practice disk protected. At `OK`, type `NEW` and press Return to clear the practice program from RAM. Open PORT.bas in **Develop → BASIC Development...**. If your browser shows the five lines as plain text, you already have the source. In the editor choose **Open...** and select **PORT.bas** in this book's **examples** folder, or use a copy you saved in Downloads. Here is its whole source:

```
10 WIDTH 40
20 PRINT "WELCOME BACK TO PORT ZERO."
30 INPUT "CALL SIGN";N$
40 PRINT "GOOD TO HAVE YOU ABOARD, ";N$;"!"
50 END
```

Line 10 asks for the wider screen. Line 30 waits for the call sign and keeps it in the string variable `N$`. Click **Run** in the **BASIC Development** window. In the CoCo display, type **NOVA** at the call-sign prompt and press Return. This time the editor put the program directly into CoCo RAM. You reached Port Zero without reading FD0.

Find the copy that matters

Where to look	What you have	When you'd use it
The FD0 row in Floppy Disks	**Port Zero.dsk**, a whole disk image on the Mac	To mount the disk or make a copy in Finder
`DIR` inside the CoCo	**PORT BAS**, one BASIC program on that disk	To bring it into RAM with `LOAD"PORT"`
BASIC Development on the Mac	**PORT.bas**, the five numbered lines as editable text	To change the source and send it to the CoCo with **Run**

That last greeting, NOVA, came from the program in RAM. Saving the editable Mac source and saving a program *on the guest disk* are two different operations. Keeping FD0 protected lets you experiment with the editor while preserving the floppy.

Want to catch the program at a particular line? Try Catch the computer in the act in the first book. You can also reach the C and Assembly workspaces from **Develop**. If your next stop is a disk of your own, go straight to Visit a disk and come home.

Color Computer Disk BASIC Quick Reference Guide. InterTAN Canada Ltd., 1987, pp. 3–9. Directory, load and save commands, and disk errors.

Trace the Mac source and the program in guest RAM.

Tape, cartridges and the slot switch

Chapter 3 of 6

Think of two old ways to start a program. With a tape, you could press Play and wait for BASIC to find a name. With a cartridge, the title could already be on the screen by the time your hands reached the keys. Both habits work here. Start with a cassette that comes with the Disk BASIC setup. You can do the slot-switch experiment at the end with just the floppy controller.

Cue up the cassette

Open **Media → Cassette...**. Look for **Hello World.cas**, the starter cassette in the **Disk BASIC** system profile. If you use a different tape in your own setup, save any recording you want to keep before changing media or loading the starter profile. The next steps replace the BASIC program in RAM, so save it first if you want it back.

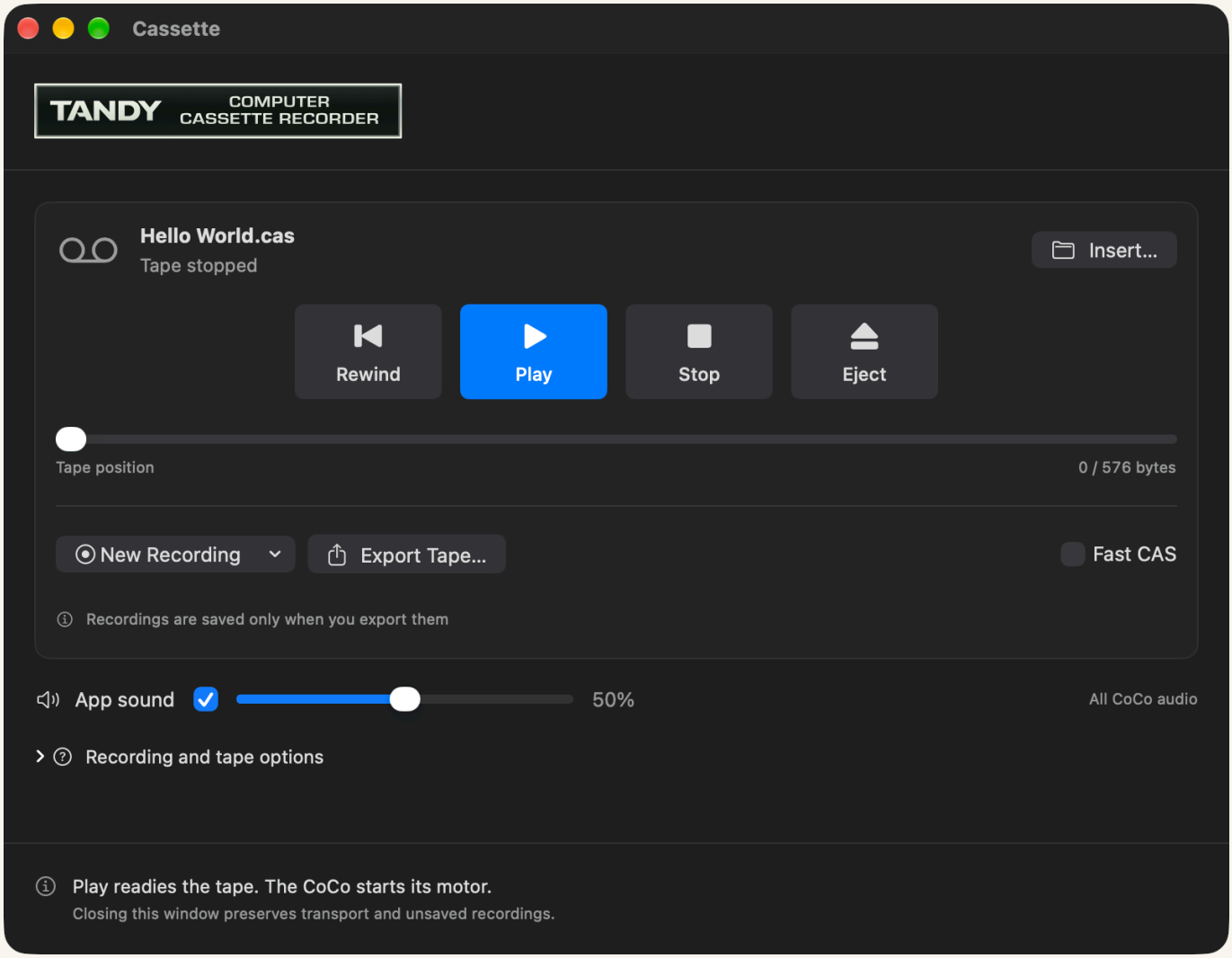


Figure 4: The current Cassette window shows Hello World.cas and its transport and app-sound controls. Use Rewind and then Play to prepare the tape for BASIC.

1. In the Cassette window, choose **Rewind** and then **Play**. Play gets the recorder ready. The CoCo controls the motor when it wants to read.

2. Return to the emulator. If it is paused, click **Run** on the toolbar. Focus the CoCo display, type `CLOAD` at `OK`, and press Return. Watch for `F HELLO`. The recorder found a BASIC program named HELLO on the tape. Wait until `OK` returns.
3. Type `RUN` and press Return. At **What planet do you live on?**, press **2**. One keystroke takes you to Mars. Look for **Hello Cold Red Dot World!** and the BASIC prompt.

If you reach the end of the tape before finding HELLO, check the filename in the Cassette window. Rewind before making another pass. You can close the window while the tape is in use. To stop the transport, use **Stop** in the Cassette window.

Power up with a cartridge

A program cartridge carries its own ROM. On a CoCo with a Multi-Pak, the switch selects the slot the machine starts from. In a Disk BASIC setup, the floppy controller already occupies slot 4. Replacing it with a program cartridge temporarily gives that startup slot a different job.

This optional walk-through uses **Color Scripsit II** for the CoCo. Use your own copy of the 1986 Tandy cartridge image if you have one. Otherwise, go straight to Watch the Multi-Pak change its mind below.

1. Save any guest program or media changes you want to keep. Open **Machine → Configuration Profiles...** and use **Save Current As...** to give your present disk setup a name. That name remembers your hardware settings and media locations. Save your BASIC program separately if you want to keep the contents of RAM.
2. In **Cartridge → Cartridges...**, find slot 4 of the Multi-Pak. Choose **Device → ROM Cartridge...** on its row and select the Color Scripsit II ROM file. Keep **Startup slot** at **Slot 4**. Confirm the restart if prompted. Press **Run** if paused.
3. Look for the **COLOR SCRIPSIT II** menu. Focus the CoCo screen. Press **E** to enter the editor, then **I** for insert mode. Type a few words. Press **Esc** to leave insert mode, then **Control-Esc** to return to the menu. Other cartridges have their own opening screens and keys.
4. In **Machine → Configuration Profiles...**, select the user profile you saved in step 1. Choose **Load...**, confirm **Load Profile**, and press **Run** if paused. The floppy controller is back in slot 4. A new BASIC prompt is ready for you.

The name on the slot matters. Disk BASIC was there because the selected controller supplied the disk ROM. Once you put Scripsit in that position, the same machine selected the word processor's ROM instead.

Watch the Multi-Pak change its mind

Look at the bottom of **Cartridge → Cartridges...** in the ordinary **Disk BASIC** setup. **Startup slot** is the selector that tells the machine where to begin. **Live routing** shows two choices the running software can make. **ROM** tells you which slot supplies cartridge program bytes. **I/O ports** tells you which slot answers device requests.

Try the two commands below only in the default Disk BASIC practice setup with the FD-502 in slot 4 and the other slots empty. Finish any disk operations first. At the start, the selector, ROM and I/O should all show slot 4. If they differ, load the Disk BASIC profile before trying this.

At the idle BASIC prompt, type the first command and press Return:

```
POKE &HFF7F,&H32
```

Look at **Live routing**. **ROM** still reads **Slot 4**. **I/O ports** now reads **Slot 3**. The startup selector stayed at 4. You just asked the running CoCo to send device requests to an empty slot while keeping the BASIC ROM in reach.

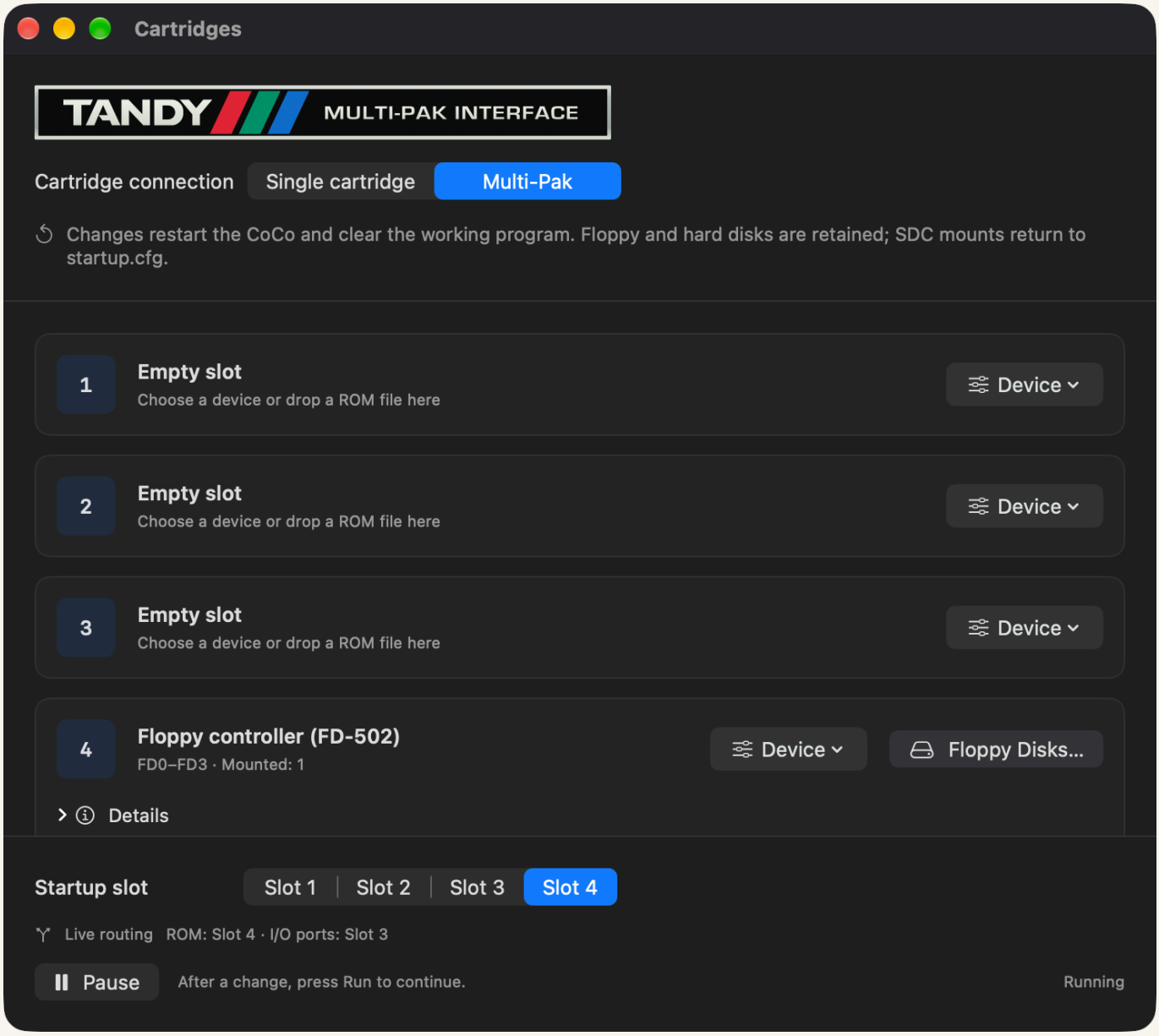


Figure 5: The startup selector highlights Slot 4. Live routing shows ROM on slot 4 and I/O ports on slot 3.

Before using any disk command, focus the CoCo again and put I/O back on the floppy controller:

```
POKE &HFF7F,&H33
```

Both live indicators should now say **Slot 4** again. The number after `POKE` changed what the Multi-Pak connected while the machine was running. You never moved its startup switch. For a closer look at the bits behind the live indicators, follow the write to a device register.

There is a second way a perfectly good drive can seem to go missing. On some setups, disk number zero belongs to a different device altogether. Which drive zero? lets you try the same directory request at three different BASIC desks.

Which drive zero?

Chapter 4 of 6

Ask Disk BASIC for a directory and the little light on FD0 flickers. Ask Hard Drive BASIC for the same directory and HD0 can answer instead. Both can show a program called HELLO. How would you tell which copy you were about to change?

Choose a setup for the software

What you remember using	Profile to open	Where to look for files
An ordinary BASIC floppy or a cassette	**Disk BASIC**	FD0 for the first floppy, or the cassette recorder for a tape
A hard disk that BASIC sees as a collection of numbered disks	**Hard Drive BASIC**	A numbered RGB-DOS volume on HD0
Disk images kept on an SD card	**CoCo SDC**	An image in the Mac folder serving as the SD card
A command shell with directories such as CMDS	**NitrOS-9 L2 V3.3.0**	The OS-9 system drive and its mounted filesystems

Pick the row that matches your disk, or try all four with starter media. Save any program you want from RAM before switching. In **Machine → Configuration Profiles…**, select the system profile, choose **Load…**, read the restart message, then choose **Load Profile**. If the machine shows **Paused**, click **Run**. A profile can bring back the same disk files, including your changes to them. Loading a profile clears the program in RAM.

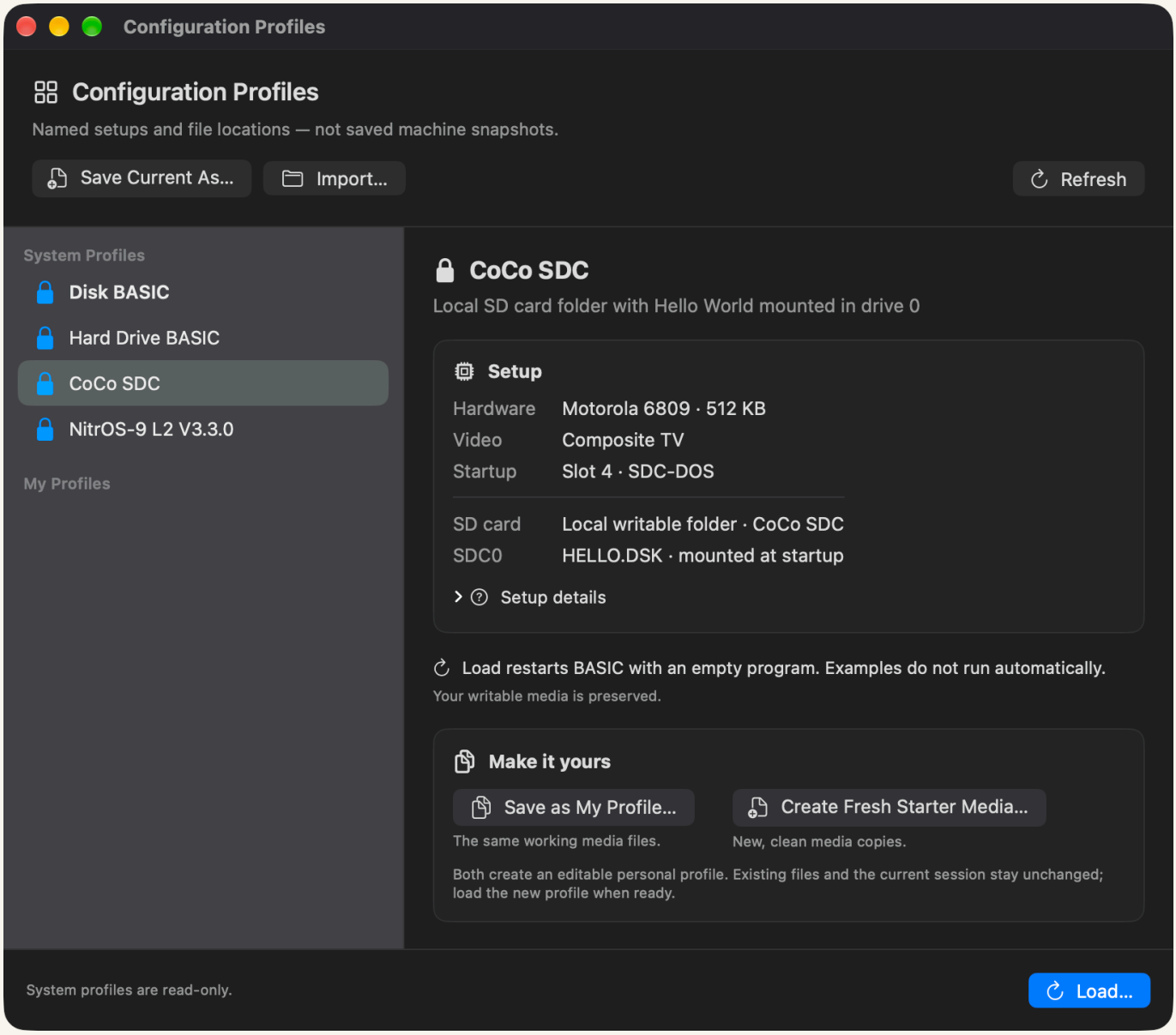


Figure 6: CoCo SDC selected among the four system profiles in Configuration Profiles. Read the description before choosing Load.

Find the floppy hiding behind the hard disk

In **Disk BASIC**, look at the Mac filename beside FD0 and then type `DIR` at the CoCo. The supplied **Hello World.dsk** contains HELLO. If you attached the chapter 2 practice disk, its directory contains PORT. A familiar filename in the CoCo listing can be reassuring, but it's the FD0 assignment that identifies the Mac file.

Now load **Hard Drive BASIC**. Keep the System view visible so you can see the drive lights. Type each command separately and watch *as you press Return*. The read can be over by the time you finish reading the directory.

Type	Watch for
`DIR 0`	HD0 answers. You have opened BASIC volume 0 on the hard disk.
`DRIVE OFF` then `DIR 0`	FD0 answers. BASIC can now see the floppy in drive 0.
`DRIVE ON` then `DIR 0`	HD0 answers again.

On fresh starter media, HELLO appears in **all three** listings. The experiment works because there are two copies of HELLO. If you keep your own files on those disks, your directory listings can look different. Before saving a program, check where you last pointed RGB-DOS.

Open an image on the SD card

Load **CoCo SDC**. Imagine a disk image lying inside an SD card's folder. Choosing that image for **SDC0** is the equivalent of putting a floppy in the first drive. The starter card's configuration already chooses **HELLO.DSK**.

1. At the BASIC prompt, enter `DIR 0`. Find HELLO.
2. Enter `LOAD "HELLO:0"`, then `RUN`. At the planet question, press **1**. You should see **Hello Nice Blue Dot World!** before BASIC returns to its prompt.
3. To select the starter image again, enter `DRIVE 0,"/HELLO.DSK"`. The slash starts the filename at the top of the SD-card folder. Type `DIR 0` to see the files inside the selected image.

Notice the two names in those commands. **HELLO.DSK** is a disk that SDC-DOS can mount. **HELLO** is the BASIC program that can run once that disk is mounted. When you add an image of your own, open the SD-card folder in Finder from the cartridge settings.

Visit the OS-9 side of the disk

Load **NitrOS-9 L2 V3.3.0** and let it boot. Wait for **Shell+** and the prompt ending in `/DD:`. You're talking to the OS-9 shell now. Try these two commands, pressing Return after each one.

```
dir /dd
dir /h1
```

The first directory includes **CMDS**, **DOCS** and **GAMES** on the system disk. The second command prints **Directory of /h1**. On a fresh setup the heading is followed by the next prompt, with room for files you might put there later.

The disk in HD1 here is the same working disk that Hard Drive BASIC uses as HD0. Its OS-9 files occupy a separate part of that disk. HELLO belongs to the BASIC side, so an empty OS-9 directory makes sense. To stay at this prompt and try something useful, find the disk behind each name, then make a personal practice setup and leave a captain's log. You can also leave this tour here. For another storage exercise, return to **Disk BASIC** through Configuration Profiles.



Figure 7: Shell+ lists folders on /dd. The next directory heading, /h1, is followed directly by the prompt on the fresh shared disk.

Same bytes, different monitor

Chapter 5 of 6

Did a game you remember in orange and blue once turn up on another screen in black and white? You can make that happen on purpose, with a program small enough to read in one glance.

Draw a picture that can fool a TV

Begin with **Disk BASIC** at `OK`. Save anything you want to keep from RAM. Open STRIPES.bas in **Develop → BASIC Development...** and choose the editor's **Run**. Confirm replacement of the current BASIC program if asked. Wait for two tall panels to finish drawing.

```
10 PMODE 4,1:PCLS 0:SCREEN 1,1
20 FOR X=24 TO 118 STEP 2
30 LINE (X,48)-(X,144),PSET
40 LINE (X+113,48)-(X+113,144),PSET
50 NEXT X
60 IF INKEY$="" THEN 60
70 SCREEN 0,0:END
```

Lines 30 and 40 both ask BASIC to draw bright vertical lines against the dark background. The loop skips a pixel between lines. There's one trick: the right panel starts **113 pixels** farther across the screen. That's an odd number. Where one pattern goes bright, dark, bright, the other starts on the opposite beat.

Line 60 waits for a key. Leave it waiting so you can change monitors with the drawing still on screen.

Try the three signals

In the main VCC Plus window, open **Settings → Video Signal** and choose **Composite TV**. The left panel looks orange and the right looks blue. On a composite connection the TV interprets picture detail and color from one signal. Fine alternating lines can produce a color that the program never asked BASIC to draw. That's *artifact color*. CoCo game artists could use it deliberately.

Choose **RGB Monitor**. The CoCo 3 can send the monitor separate red, green and blue signals. Look closely at the panels. Both show thin black and white lines. The program is still waiting at line 60.

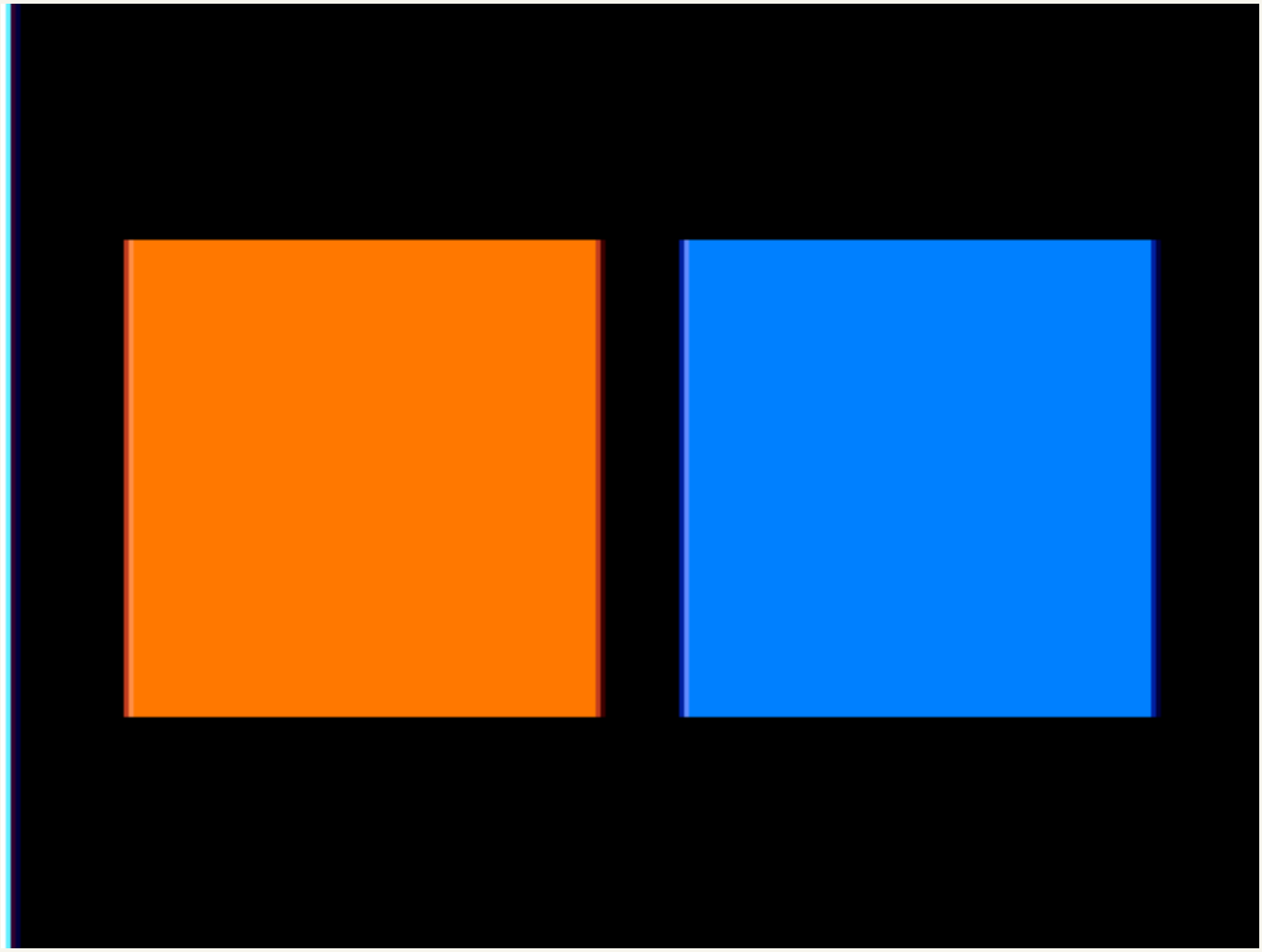


Figure 8: Composite TV. The left panel looks solid orange and the right panel looks solid blue.

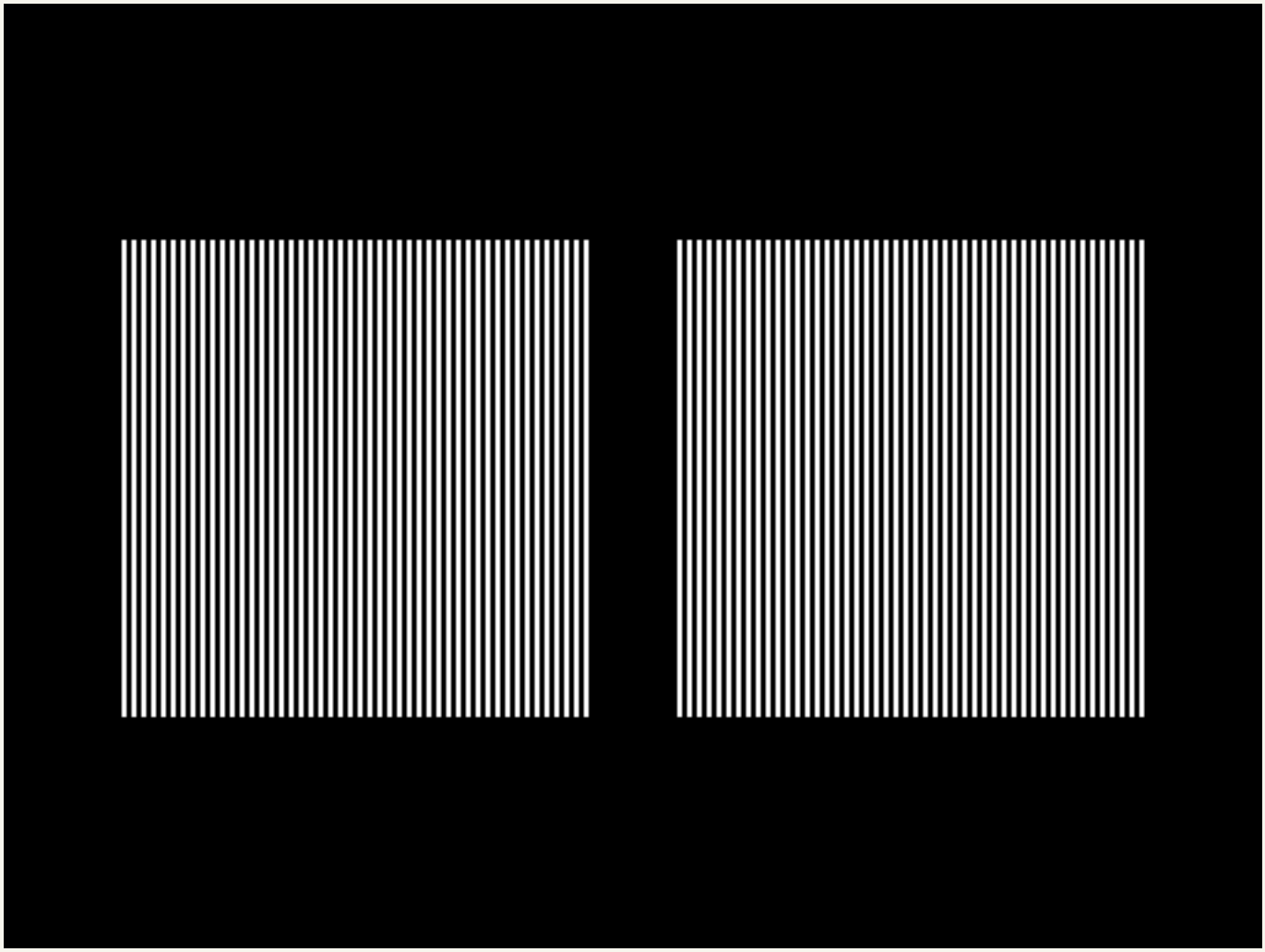


Figure 9: RGB Monitor. The same panels show alternating light and dark vertical lines from top to bottom.

Try **Composite TV – Alternate Colors**. The solid-looking panels are back, with blue on the left and orange on the right. If a favorite game's colors look reversed, this is a setting worth trying. For a CoCo 3 program designed for an RGB monitor, try **RGB Monitor** first.

Choose the signal you want to keep. Use **Input** → **Focus CoCo Display** and press **A**. The program switches back to text and returns to `OK`.

Find the joystick port a game reads

An old game can put its opening screen on the monitor and then wait for a joystick on one particular port. Here's a way to see what BASIC receives before you start the game.

1. Make sure your source is saved. Start with a fresh Disk BASIC program, or type `NEW` at `OK` to clear the practice program from RAM.
2. Enter these two numbered lines and the run command, pressing Return after each one.

```
10 PRINT@0,JOYSTK(0);JOYSTK(2)
20 GOTO 10
RUN
```

3. In **Input** → **Keyboard and Joysticks…**, find the **Right CoCo joystick** and **Left CoCo joystick** assignments. If you have a controller mapped, use that stick. To try keyboard control, note your previous

setting, choose **Keyboard** for one port, and select **Arrows + Z / X** in its options. Put focus back on the CoCo.

4. Move the stick from left to right, or hold and release the left and right arrow keys. Watch the two numbers at the upper left. The **first** follows the right CoCo port; the **second** follows the left one. With centered controls you should see about **31 31**. With only the right-hand port moving, try for **0 31** and **63 31**.

If the other number changes, you've just found the port your input actually reaches. Use the input window to match that port to the one your game expects. Press **Esc** to stop the loop. Restore any input assignment you changed for this experiment.

Visit a disk and come home

Chapter 6 of 6

Suppose a friend gives you an image of a BASIC disk. You want to try it, keep the disk safe, and get your usual floppy back afterward. Make the round trip once with Port Zero and you'll have a routine for your own collection.

Give your starting setup a name

Start in **Disk BASIC** with the ordinary starter floppy in FD0. Open **Media → Floppy Disks…** and read the filename in that first row. With the supplied starter it reads **Hello World.dsk**. Write down the name so you can recognize it on your return.

If you have a BASIC program in RAM that you want to keep, save it to your own disk or tape first. A configuration profile remembers things such as your machine settings, the cartridges in their slots, and which Mac files your drives use. Your program in RAM needs a save of its own.

Open **Machine → Configuration Profiles…**, choose **Save Current As…**, and call your profile **Home Port Setup**. Save the name. You have now recorded the setup with the starter floppy attached.

Take a protected copy for a spin

1. In Finder, duplicate your friend's image in a writable folder. For this trip you can use the Port Zero practice disk and make a duplicate just as you would for a treasured disk.
2. In **Media → Floppy Disks…**, choose **Replace…** beside FD0 and select the duplicate. Expand **Disk options** and turn on **Write protected**. Check for your duplicate's filename and the words **Write protected** on the *same drive row*. Compare it with the protected drive in chapter 2.
3. Click **Run** if the emulator shows **Paused**. Give the CoCo display keyboard focus. At `OK`, enter `DIR`. For Port Zero, look for PORT, then enter `LOAD"PORT"` and `RUN`. When asked for a call sign, type **ACE** and press Return. The CoCo welcomes you aboard.
4. On a personal disk, use its own instructions to start the program. If all you know about that disk so far is its directory, you've still learned something useful. You can come home with the original safe and the names of its files in hand.

Return to your starter floppy

Open **Machine → Configuration Profiles…**, select **Home Port Setup**, and choose **Load…**. Read the confirmation and choose **Load Profile**. If VCC Plus asks you to resolve changes to a temporary disk or an unsaved recording, keep the data you want before switching. Leave a discard action for data you're ready to lose.

Check **Media → Floppy Disks…** again. The filename you wrote down should be back, with **Saves automatically** under it. Loading the profile restored the drive assignment and its settings.

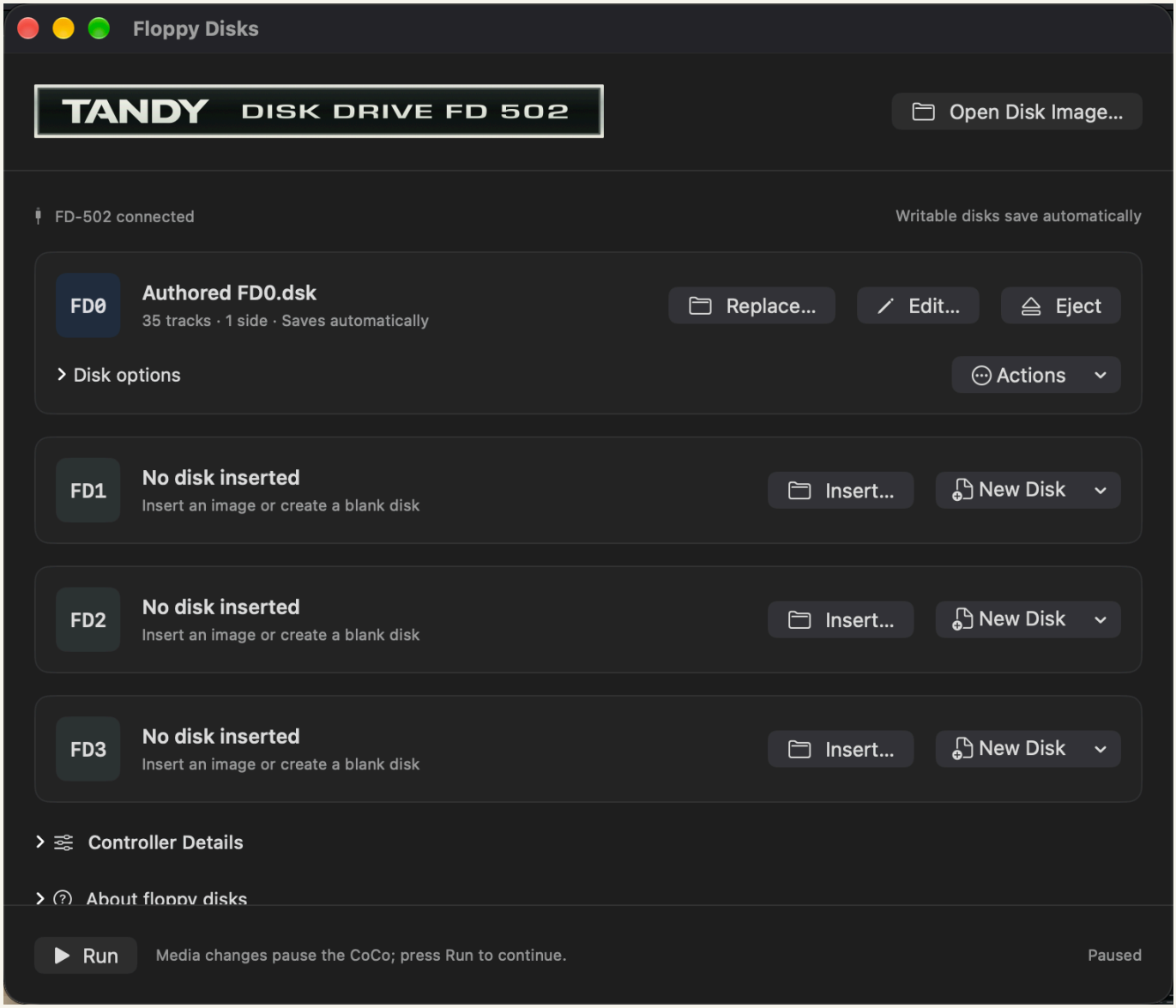


Figure 10: The current FD0 row shows a representative authored disk with Saves automatically. In your restored Home Port Setup, look for Hello World.dsk and open Disk options to confirm Write protected is off.

Click **Run** to resume the restored setup. Type `DIR` at the fresh BASIC prompt. On the starter floppy you'll find HELLO again. You brought your equipment and disks home. To bring back a BASIC program, load the copy you saved.

Come home after a Software Library visit

If the **Software Library** set up the machine for a title you launched, reopen **Media → Software Library...**. Look near the window title for the curved back arrow, **Return to Previous Setup...**. Read the confirmation and choose **Restore Setup** when you're ready to leave the title.

That arrow belongs to a Library visit. For a disk you inserted yourself, **Machine → Configuration Profiles...** holds the named setup you saved. Both routes restore equipment and file assignments. A game score or the place where a running program paused survives a restart only if the program saved it somewhere you can load again.

Beside the keyboard

Appendix A

Keep this page open next to the emulator if you're trying a personal disk for the first time. The practice files also work offline. Save a copy in a Finder folder you can return to.

Pick up the practice files

File	Open it here	What you can do
Port Zero.dsk	**Media → Floppy Disks...**, in FD0	A whole BASIC floppy with PORT already on it. Follow the protected-disk exercise.
PORT.bas	**Develop → BASIC Development... → Open...**	The same program, in five editable lines. It asks for your call sign on a 40-column screen.
STRIPES.bas	**Develop → BASIC Development... → Open...**	Run the program and leave it waiting while you change video signals.

In the app, select the disk link and choose where to save a working copy. Select either BASIC link to open an editable example in BASIC Development. Some browsers open a BASIC listing as plain text. To use the offline original, choose **Open...** in BASIC Development and select the named file in the **examples** folder beside this book. Make a copy before changing it. Port Zero needs the floppy controller and Disk BASIC. The two listings run from the editor at a BASIC prompt. Open the *disk image* when you want the drive to take part.

When an old habit gets a surprising result

What you see	What to check
Typing moves a Mac control.	Click the CoCo display or use **Input → Focus CoCo Display**. If the toolbar says **Run**, resume the machine first.
You clicked Run but BASIC is still at OK.	The Mac toolbar resumed the machine. To start a BASIC program already in RAM, type `RUN` at the prompt.
You can read PORT, but trying to save LOG gives `?WP ERROR`.	On the protected practice disk that's the expected result. Run PORT again from RAM. To keep a change, use your own writable copy.
You recognize the filename HELLO but suspect you're reading the wrong disk.	Check your profile. In Hard Drive BASIC, compare the hard disk and floppy. Both can have a HELLO at drive zero.
You came back to the right disk but your program is gone from the screen.	A profile reopens its media and starts a fresh machine. Load the copy you previously saved on that disk. See Come home.
You expected blue and orange but see two sets of stripes.	Keep STRIPES running and choose **Composite TV** under **Video Signal**. Compare the two actual pictures.

Before you put a treasured disk away

Look at the drive's Mac filename, the write mode beneath it, and any message about pending changes. **Write protected** prevents guest writes. **Saves automatically** writes changes to that Mac image. **Temporary changes** holds writes in memory. If you want to keep temporary edits, choose **Actions → Save As...** and give the copy its own name before ejecting or switching setups.

Try it on disposable media in Prove which copy changed.

For another turn at the desk, keep both pieces: the file containing your program and a note or profile telling you how you ran it. The book's final exercise walks through saving your setup and returning to it.

Behind the desk

Appendix B

When you change the Multi-Pak's routing in chapter 3, you can watch the two slot numbers change on the Mac. On the original hardware you could see the switch but had to work out where the signals went. These guides put the hardware side of the experiment within reach.

Historical reading

- Tandy Corporation. *Introducing Your Color Computer 3*. 1986. The opening hardware tour introduces the keyboard computer, its connections, and the displays an owner might use.
- Tandy Corporation. *TRS-80 Multi-Pak Interface Owner's Manual*. 1983. Read the equipment setup and slot-selector instructions alongside the slot experiment.
- Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17, and functions, pp. 18–22. A pocket companion for reading the listings and identifying BASIC statements and functions.
- *Color Computer Disk BASIC Quick Reference Guide*. InterTAN Canada Ltd., 1987. Disk commands, pp. 3–8, and errors, p. 9. Useful beside Port Zero.
- Lindner, Tim, producer. *CoCo SDC*. Fourth version, February 2023. Hardware and software by Darren Atkinson. The hardware overview and command summary give the physical-card background for the profile chooser.
- CoCopedia contributors. "Color Computer 3." *CoCopedia*, revision 11134. A community history of the machine, its graphics and its place in the Color Computer family.

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

How we made the pictures

The FD-502 on the cover uses the original VCC Plus hardware illustration. The Mac windows were captured in VCC Plus. The two stripe panels came from one running program, after changing the video signal and leaving its drawing alone. Artwork credits.

Pick your next project

If you once saved a BASIC program but never stepped through one, Catch the computer in the act walks through stopping a program and watching a variable change. If someone at your desk has never used a CoCo, Meet Your Color Computer gives you a project to do together.

For a longer voyage, write a log that survives a restart. For a command while you work, open **Develop → BASIC Quick Reference**. For another disk of your own, keep the field guide beside the emulator.

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.

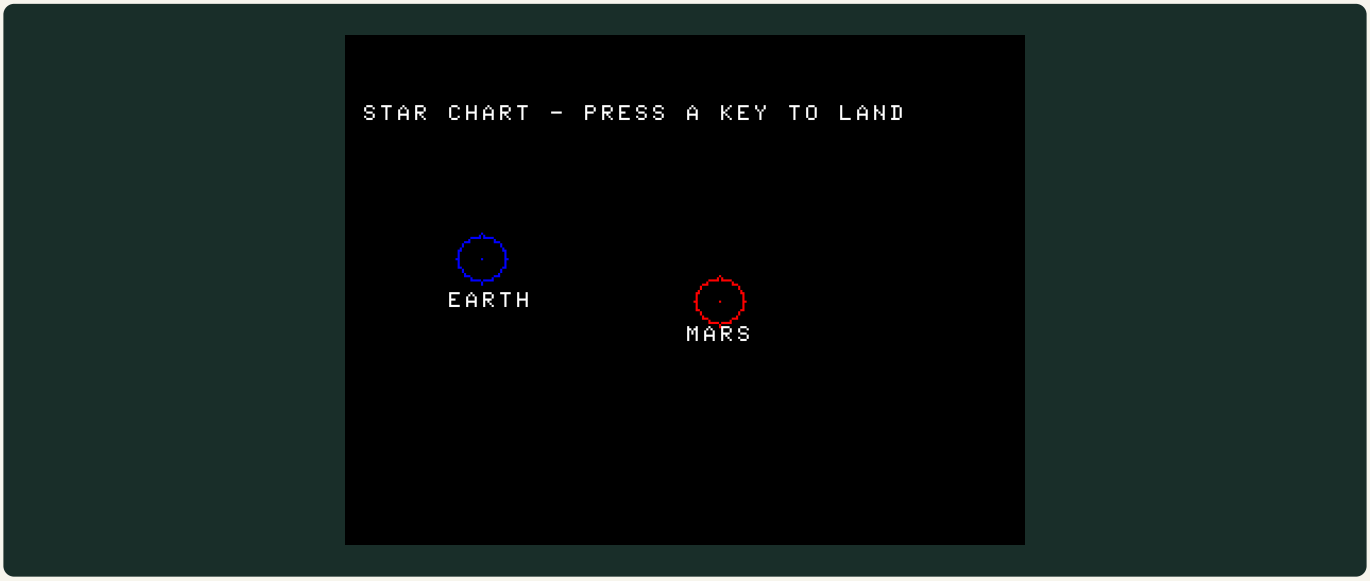
The CoCo 3 and CM-8 monitor presentation in VCC Plus's skeuomorphic System view was inspired by SockMaster's "pic of CoCo 3 and CM-8 monitor," published on CoCopedia. VCC Plus's artwork was designed and rendered independently. The remaining hardware illustrations were independently designed using historical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

Read this book online

CMOC license notice · LWTools COPYING notice

Programming in CoCo BASIC on the Color Computer 3

Earth or Mars? Build a flight deck, hunt down a supply leak, and leave a log that the next crew can read after you shut down the CoCo.



Planet Explorer running on a CoCo 3, with Earth in blue and Mars in red.

Brian K. Holman

Book 3 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Make the CoCo answer . . . 3
Choose a destination . . . 5
Count the supplies . . . 7
Find the missing supply . . . 9
Give a chart its names . . . 13
Give a routine a job . . . 15
Design a flight deck . . . 17
Draw a planet you can find again . . . 19
Listen for a signal . . . 24
Keep a log past bedtime . . . 26
Take the voyage to another machine . . . 30
Fly, land, power down, remember . . . 32
BASIC at your elbow . . . 34
Where BASIC meets the machine . . . 72
What did that error mean? . . . 74
Publication and artwork notices . . . 79

Make the CoCo answer

Chapter 1 of 12

Your flight computer is awake. Before you let it steer, make it show you how it thinks. You can ask three questions with almost the same keys and get three different answers.

Try the keyboard at OK

Start with **Disk BASIC** in a practice session. If you have a program in the CoCo that you want to keep, save it before using the numbered lines below. Click inside the CoCo display. When you see `OK`, type each of these lines and press **Return** after each one. Predict the next answer before you press the key.

```
PRINT 2+3*4
PRINT (2+3)*4
PRINT "2+3*4"
```

The first answer is **14**. The star means multiply. BASIC multiplies 3 by 4 first, then adds 2. The second answer is **20**. Parentheses tell BASIC to finish the addition first. The third answer is `2+3*4`, right down to the star. Quotation marks tell BASIC to take those five characters as **text**. Programmers call a piece of text a *string*.

An instruction you type this way is an **immediate command**. BASIC carries it out as soon as you press Return. Use the Mac keyboard as the CoCo keyboard while the display has focus. If typing moves a Mac control instead, click the CoCo display again.

Give tomorrow’s pilot a message

At the practice prompt, enter `NEW` to clear the old listing from guest memory. Type these two lines, pressing Return after each.

```
20 PRINT "PLANET EXPLORER"
30 PRINT "1 EARTH   2 MARS"
```

What happened to the menu? You saw your typing, then a new input line. The numbers tell BASIC to store the instructions. Type `LIST` to see what it kept. Now type `RUN`. This time the CoCo prints the title, prints the two choices, and says `OK`. You can type `RUN` again any time you want that menu back.

Slip a line between the two you already have. Enter `25 PRINT "WELCOME ABOARD"`, then `RUN`. The welcome appears between the title and the destinations. BASIC follows the **numbers**, even though you typed line 25 last. Enter just `25` and press Return. Type `LIST` again. Lines 20 and 30 remain. Entering a bare line number at the guest prompt deletes that line. An entire replacement line with the same number changes it. This is why you often see gaps between BASIC line numbers.

Keep a Mac copy

Open **Develop → BASIC Development…**. Choose **Read from CoCo**. If the editor is holding work you want, use its save option first. Look for the same two numbered lines you saw with LIST. Choose **Save As…** in the editor’s file actions and give your Mac source a name such as **Menu.bas**.

Place	What lives there	How to find it
CoCo RAM	Instructions the guest BASIC interpreter can run, plus working values while it runs	LIST at the guest prompt shows the instructions.

Place	What lives there	How to find it
Mac editor	Editable text, including any changes you have typed since the last transfer	Read from CoCo pulls the current guest listing into the editor. Save keeps the text in a Mac file.

Load into CoCo sends the editor's listing to guest RAM and leaves BASIC ready. The editor's **Run** sends it and starts it. Either operation can replace the program in guest memory; read the replacement prompt before choosing. The **Run** control in the main emulator window resumes the entire machine after a pause. Use the editor's Run when you want the BASIC listing to start at the beginning.

In the next lesson, you'll add the missing part of the menu: the CoCo will wait for a passenger to choose a planet.

Choose a destination

Chapter 2 of 12

A good pilot can change their mind. A good menu can survive a slip of the finger. Let's type the wrong answer on purpose and see whether the flight computer gives us another chance.

Fly the menu

Open HELM.bas from the Manual Library. It opens an editable example. In a browser, save the file and open it with **Develop → BASIC Development... → Open...**. Choose the editor's **Run**. If you want to replace the practice listing from chapter 1, confirm. Click the CoCo display.

1. At **SET COURSE**, enter **X** and press Return. Look for **TRY 1 OR 2** and another question.
2. Enter **1** and press Return. Look for **BOUND FOR EARTH** and `OK`.
3. Run again and enter **2**. Now you're **BOUND FOR MARS**.

Here is the entire helm. You already know the two lines that print the menu.

```
10 WIDTH 40:CLS
20 PRINT "PLANET EXPLORER"
30 PRINT "1 EARTH   2 MARS"
40 INPUT "SET COURSE";C$
50 IF C$="1" THEN P$="EARTH":GOTO 90
60 IF C$="2" THEN P$="MARS":GOTO 90
70 PRINT "TRY 1 OR 2"
80 GOTO 40
90 PRINT "BOUND FOR ";P$
100 END
```

Follow the choice with your finger

`WIDTH 40` puts the CoCo 3 in its 40-column text mode. The colon joins two instructions on one numbered line. `CLS` clears the screen before the sign goes up.

On line 40, **INPUT** prints a prompt and waits for a reply followed by Return. `C$` is a *variable*: BASIC remembers a value under that name. The dollar sign says its value is a string. When you type **2**, BASIC stores the character "2". That's why the comparison on line 60 has quotes around the digit. A room number or a menu choice can be a perfectly good piece of text.

Put one finger on line 50. The equal sign in `IF C$="1"` asks a question. If the answer is true, the part after **THEN** assigns a name to `P$` and sends execution to line 90. If it's false, BASIC skips the rest of line 50 and tries line 60. The equal sign in `P$="MARS"` has another job: it puts a value into a variable. Watch for whether you're reading a question or an assignment.

Reply	Lines after INPUT	Why
1	50 → 90 → 100	The first comparison succeeds and the jump skips the rest of the menu.
2	50 → 60 → 90 → 100	The second comparison succeeds.
X	50 → 60 → 70 → 80 → 40	Both comparisons fail. The program prints a hint and jumps back to INPUT.

The semicolon in line 90 prints two things side by side. The space at the end of the quoted words leaves room before the planet name. An extra menu trip costs almost nothing when all the program has to do is jump back a few lines.

To try the same decision in another language, give Mars a key in C. That version reads one keystroke as soon as it arrives. To see where the processor keeps the chosen color and message, catch the two assembly routes where they meet.

Ask a question about the answer

After the Mars run, stay at the guest's `OK` prompt. Your listing has ended, but `P$` still holds the name from that flight. Try these immediate commands one at a time.

```
PRINT LEN(P$)
PRINT LEFT$(P$,1)
PRINT RIGHT$(P$,2)
```

You should get **4**, **M**, and **RS**. A *function* takes a value and produces a result. LEN counts characters. LEFT$ and RIGHT$ give you pieces of the original string. None of those questions changed the name. In the chart-data lesson, BASIC will keep several names at the same time.

Count the supplies

Chapter 3 of 12

Lay six imaginary food packs on the table. Spend one on each of three turns. How many do you expect after turn one? After turn three? Your prediction is a perfectly good way to test a computer program.

Keep one number in working memory

Open RATIONS.bas and read the listing **before** you run it. A name without a dollar sign can hold a number. Here `S` stands for supplies and `T` stands for the current turn.

```
10 WIDTH 40:CLS
20 S=6
30 FOR T=1 TO 3
40 S=S-1
60 PRINT "TURN";T;" SUPPLIES";S
70 NEXT T
80 END
```

Read line 40 aloud as “take the old number of supplies, subtract one, and store the result back under S.” BASIC works out the expression on the right before it replaces the old value on the left. On the first trip through the line the arithmetic is six minus one. On the next trip it is five minus one. The same line can do a different calculation each time.

Take another turn

FOR sets `T` to 1 and gives the loop a last turn of 3. Line 40 spends one pack; line 60 prints the current turn and what is left. At **NEXT T**, BASIC adds one to the counter. If another turn fits, BASIC goes back to the first instruction inside the loop. Otherwise it carries on after NEXT.

Choose **Run** in the editor. Compare your pencil-and-paper answer with the guest screen.

Turn when line 60 prints	Supplies arriving at line 40	Supplies printed at line 60
1	6	5
2	5	4
3	4	3

At `OK`, type `PRINT S,T`. The CoCo prints **3** and **4** in separate fields. The **4** is a surprise worth keeping: NEXT moved the counter on one final time and tested it. Four was past the end. There was never a fourth trip through line 60. A comma in PRINT separates fields; the semicolons in the original listing keep the labels and numbers close together.

Check the arithmetic before changing a program

At the prompt, try `PRINT 6-1*3`. The answer, **3**, agrees with the final cargo count. That single calculation tells you the amount left at the end. The loop also tells a crew member how much is left *during* each turn. Try `PRINT (6-1)*3` and the answer becomes **15**. The parentheses asked the CoCo to repeat five packs three times.

Choose the size of one step

Try these two loops at the prompt. The colon lets several instructions share a line. The semicolon after J keeps the numbers on the same row.

```
FOR J=3 TO 1 STEP -1:PRINT J;:NEXT J
FOR J=2 TO 6 STEP 2:PRINT J;:NEXT J
```

The first loop prints **3 2 1**. The second prints **2 4 6**. In RATIONS, NEXT used the usual step of one. Here you chose the size and direction. The end value belongs to the loop when the step lands on it.

For another kind of number question, try `PRINT ABS(-3);INT(6.5)`. The results are **3** and **6**. ABS measures how far a number is from zero. INT takes the greatest whole number at or below its input. For other built-in number tools, the language reference gives each one's inputs and result. The next flight has an arithmetic problem you can catch with a watch.

Find the missing supply

Chapter 4 of 12

Some bugs make BASIC stop and print an error. This one is trickier. Every line is legal BASIC, the ship makes all three turns, and the crew ends up hungry.

Reproduce the fault

Open LEAK.bas and choose **Run**. You'll see 4, 2 and 0 supplies left after the three turns. In chapter 3 the same six packs lasted the whole trip with three to spare. Where did the first extra pack go?

```
10 WIDTH 40:CLS
20 S=6
30 FOR T=1 TO 3
40 S=S-1
50 S=S-1
60 PRINT "TURN";T;" SUPPLIES";S
70 NEXT T
80 END
```

BASIC is quick enough to finish the flight before you can read the first result. Use a **breakpoint** to ask the editor to wait for you at a particular instruction. That lets you investigate while the clues are still in RAM.

Stop before the subtraction

1. In the editor, click the breakpoint circle in the margin beside **50**. To set it from the keyboard, put the insertion point on that line and activate **Toggle BASIC breakpoint at insertion point**.
2. Open **Variables and Watches**. Select **Watches**, add `S`, then add `T`. The editor will fill in their values when BASIC stops.
3. Choose the **editor's Run** again. The highlight should land on line 50. Read both watches before doing anything else. **T is 1. S is already 5.** Line 40 has spent one pack. Line 60 has yet to print the first turn.
4. Choose **Step Into BASIC statement** once. The highlight moves to line 60. Read the watches again. **T is still 1. S is now 4.** One crew, one turn, two subtractions. You found the leak.

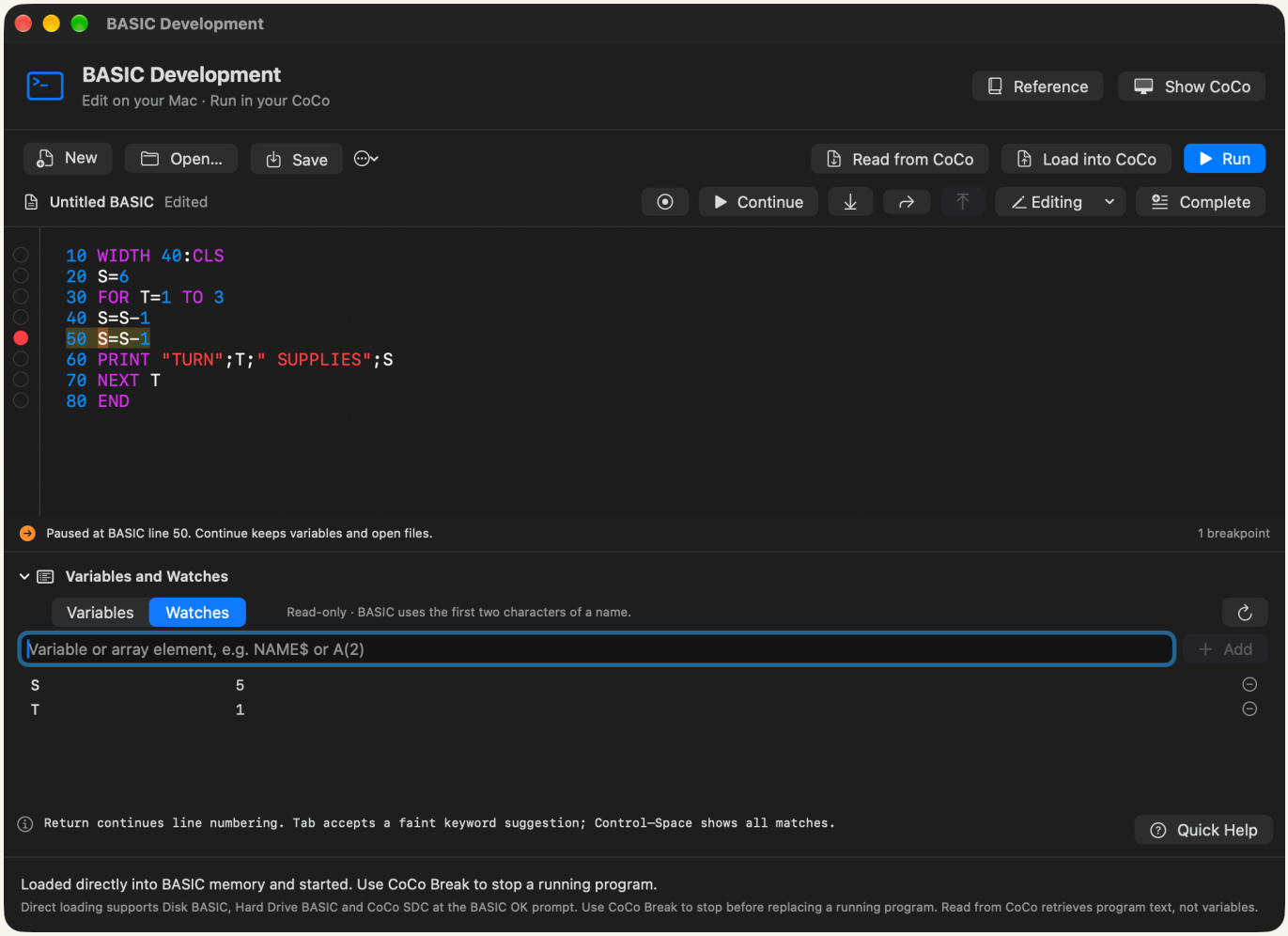


Figure 1: Before line 50 runs, the Watches list shows S at 5 and T at 1. The first turn is still in progress.

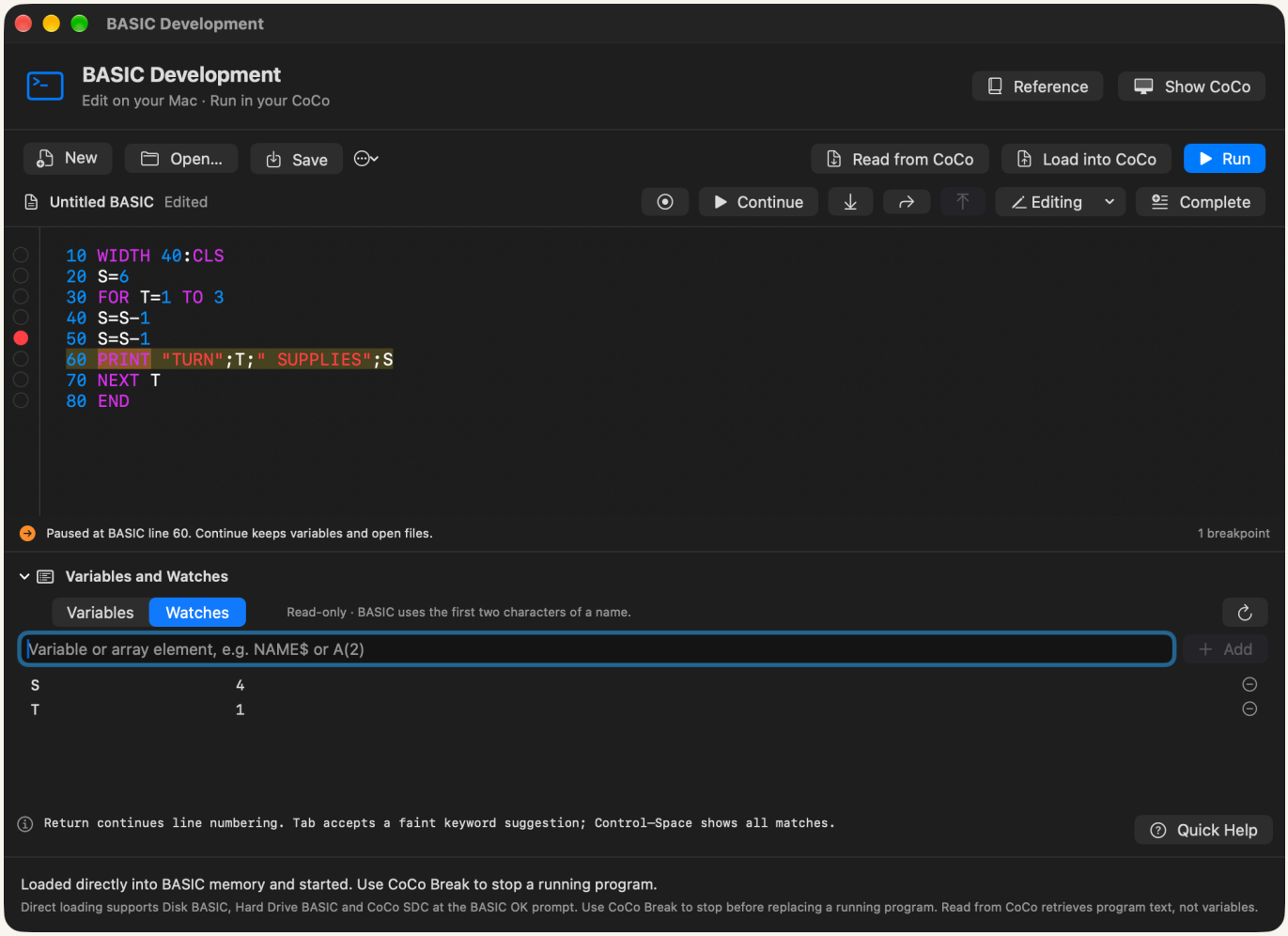


Figure 2: One step later, line 60 is highlighted. S has fallen to 4 while T is still 1.

The highlighted instruction is the next one BASIC will carry out. That small detail makes the screen and the watches agree: at the second stop S is 4, but the print instruction that will display it is still ahead.

Choose **Continue**. Line 60 prints the first result, NEXT moves T on, and the breakpoint catches the next trip through line 50. Choose Continue twice more to finish the three turns. You can use the **Variables** tab to inspect everything BASIC has created. A watch lets you pick a few names and keep your attention on them.

Repair the listing

Finish the run before editing. Click the breakpoint circle again to remove it. In your editable example, delete the entire line `50 S=S-1`. Keep line 40. Run the edited copy. You should see 5, 4 and 3. RATIONS.bas is the supplied repaired version if you want to compare. The gap between 40 and 60 costs the CoCo nothing.

Three kinds of stop. A breakpoint belongs to this editor document. **Step Into** carries out one BASIC statement. **Continue** runs until another stop or the end. BASIC also has a STOP statement and a CONT command of its own. A new run or a changed program can make guest continuation impossible. When BASIC reports a short error code and line number, use **Show Line** in the editor to inspect that statement and the error guide to look up the code. TRON and TROFF are the older guest tools for printing a trace of line numbers as BASIC runs.

When a step waits for you

A BASIC statement that asks for input can still be running while you look at the editor. **Pause BASIC step** appears during a pending step. A mid-statement pause can have no highlighted source statement because the

interpreter has yet to reach the next one. Choose **Continue** and focus the CoCo display to answer the waiting program. The debugger can step from a statement boundary again when it reaches a BASIC breakpoint.

A runtime error leads somewhere different. In the error banner, **Show Line** takes you to the numbered statement. The question-mark button, **Explain this BASIC error**, opens its entry in the local Quick Reference. The failed run has ended. Correct the source and run again when you are ready. To remove the document's stops together, use **Editing → Clear All Breakpoints**.

Move a line and keep its address

Line numbers are also destinations. In HELM, the bad-reply path jumps to 40 for another question. If you changed that label by hand and forgot the jump, the next bad reply would go looking for the old number.

For a small experiment, open HELM.bas again and run it. Enter **2** to reach the guest prompt. Type `RENUM 100,0,10`, then `LIST`. You asked BASIC to start the new numbers at 100, begin with the old lines from 0 onward, and leave a gap of ten. The old line 80 now reads **170 GOTO 130**. The question has moved to 130, and the jump followed it. Enter `RUN` and **1** to try the renumbered menu. Look for **BOUND FOR EARTH**.

That change happened in guest RAM. To bring it back to the Mac, use **Read from CoCo**. To renumber an *editor copy* instead, use **Editing → Renumber…**, choose the first number and increment, and apply the change. **⌘Z** undoes that edit before transfer. This distinction lets you try out new numbering on one side while keeping a saved copy on the other.

Give a chart its names

Chapter 5 of 12

What could go wrong if the computer drew Earth at the right place but printed “Mars” underneath? A chart needs more than a bag of names and numbers. It needs a way to keep the pieces of each planet together.

Make two cards for your chart

Open CHART.bas. Begin at the bottom of the listing, where its cartographer left this row of data.

```
150 DATA EARTH,64,80,MARS,176,100
```

Read three items at a time: a name, a distance across the picture, and a distance down. Make two little cards on paper.

Card number	Planet name	X, across	Y, down
1	EARTH	64	80
2	MARS	176	100

Which planet will appear farther to the right? Which will appear lower? In this graphics mode X grows as you move right, and Y grows as you move down. Mars has the larger number in both columns.

Deal the cards into RAM

```
20 DIM P$(2),X(2),Y(2)
30 FOR I=1 TO 2:READ P$(I),X(I),Y(I):NEXT I
```

`DIM` makes room for values you can look up by number. Each group is an **array**. `P$(1)` means the name for card 1, while `P$(2)` means the name for card 2. `X(2)` and `Y(2)` belong to that second card, too. You can think of the number in parentheses as a drawer label. BASIC calls it a *subscript*. A dimension of 2 makes positions 0, 1 and 2 available. This program uses positions 1 and 2 to match the numbers on the menu.

On line 30, READ asks for the next three items in DATA order. On the first pass through the loop, I is 1; on the second pass I is 2. READ remembers how far it got through the DATA list. So `X(2)` receives 176, even though the DATA line is more than a hundred line numbers away. BASIC only needs the list and a place to put each item.

Earth’s name is a string, so the dollar sign belongs on the name array. The X and Y arrays hold numbers. That lets the drawing code do arithmetic with the coordinates. For example, line 75 divides a pixel coordinate by eight and uses INT to find a whole text-cell position for a label.

Check the real chart against your cards

Select **RGB** under **Video Signal** for the colors used in this book. Choose the editor’s **Run**. Find the labels. Compare their positions with the paper cards. Press **A** on the CoCo keyboard. Now the text screen tells you the two marker values that the program read back from the picture.

At `OK`, enter `PRINT P$(2);X(2);Y(2)`. You should get **MARS 176 100**. You just asked BASIC for all three entries belonging to one card, after the drawing vanished. Try `PRINT LEN(P$(1));LEFT$(P$(2),1)` and predict both results. Earth has five letters, and Mars begins with M, so the CoCo prints **5 M**.

What if the next crew wants to read the original DATA items again? Type these three commands at the prompt, one per Return.

```
RESTORE
READ N$,E,V
PRINT N$;E;V
```

This time the result is **EARTH 64 80**. RESTORE moved the DATA reader back to the beginning. The new READ put the first three items into three new variables. The older arrays still have their own values. When you want to draw with those coordinates yourself, open Draw a planet you can find again.

Give a routine a job

Chapter 6 of 12

Think of the last thing a pilot ought to do after landing. The next crew needs a record of the trip. What if the flight program could simply ask someone to file that record and come back when the job was done?

Leave a place to come back to

Open VOYAGE.bas. First read this little piece of the main flight plan.

```
135 GOSUB 300
140 PLAY "T180O3L8CEG"
150 GOSUB 200
160 PRINT "LOG SAVED FOR ";P$(D)
170 END
```

A **subroutine** is a group of instructions with a job of its own. `GOSUB 300` sends BASIC to the landing routine. BASIC remembers *where to come back*: when the landing routine reaches RETURN, the next instruction after the call is the PLAY at line 140. The next call, `GOSUB 200`, sends BASIC to the filing routine. Its RETURN brings us back to line 160.

Turn the two calls into words and you can read the whole flight plan: land; play a tune; file the log; tell the pilot it was saved. The filing routine happens to be earlier than the landing routine in the listing. You can read them in the order that the pilot uses them. Line 170 ends the main plan, so BASIC leaves the code below it alone until a GOSUB asks for a job.

```
200 OPEN "O",#1,"VOYAGE.DAT:0"
210 WRITE #1,P$(D),S
220 CLOSE #1
230 RETURN
```

The file number `#1` connects the open, write and close instructions. Both the main program and the routine know the destination number D, the name array P$, and the supplies S. A GOSUB in this BASIC shares its caller's variables. That makes a short routine easy to call. It also means changing S inside the routine would change what the crew sees afterward.

Step into the file room

Use **Disk BASIC** with a disposable initialized floppy mounted in **FD0**. In **Media → Floppy Disks…**, check the name of that disk and look for **Saves automatically**. VOYAGE writes a file called **VOYAGE.DAT** and replaces an earlier flight under that name. If you don't have a practice disk yet, make one before running this step.

1. In the VOYAGE editor, mark line **150** as the only breakpoint. Add **D** and **S** to Watches. Choose **Run**.
2. Choose **2** and press Return at the helm. When the Mars chart asks you to land, press **A** on the CoCo keyboard. Look at the editor when it stops on line 150. D is **2**, and S is **3**.
3. Guess which line **Step Into BASIC statement** will highlight. Try it. The highlight moves to **200**, the first instruction of the file job. The crew hasn't announced a save yet.
4. Choose **Step Out of BASIC subroutine**. The CoCo finishes the job, including CLOSE. It stops on line **160**, where the main flight plan resumes. Choose **Continue** and look for **LOG SAVED FOR MARS**.

Choose how much of a job to watch

Leave the breakpoint at 150. Fly again, this time entering **1** for Earth and pressing **A** to land. At the stop, choose **Step Over BASIC subroutine**. Predict the next highlighted line. You land on **160** again, this time without a

pause at 200. The filing job still ran. Choose **Continue** and you see **LOG SAVED FOR EARTH**. Remove the breakpoint when you finish this exercise.

Use **Step Into** to see how a job works, **Step Over** when you already trust the job, and **Step Out** when you've seen enough of the current routine. A breakpoint you deliberately set *inside* a routine can still bring Step Over to a stop there. A GOTO, unlike GOSUB, makes no promise to come back.

As your programs grow, **ON ... GOSUB** can use a number to select one of several routines. **DEF FN** gives a name to a calculation that returns a value; BASIC's own LEN and INT are familiar examples of how it feels to use a function. Look up these forms in the language reference. The flight program already has two useful jobs, and you can follow both of them all the way back to the cockpit.

Design a flight deck

Chapter 7 of 12

You have one screen and two jobs for it. At takeoff the pilot needs a short question and two big choices. After landing the crew wants to compare a name at the left with a number farther across the page. Should both screens have the same number of columns?

Walk across three screens

Open CONSOLE.bas in BASIC Development and click **Run**. Click the CoCo display. Start with your hands off the keys. Count how many times the *same* Mac window changes before you get back to the helm.

1. The first screen says **32 COLUMNS**. Tap and release A.
2. The second says **40 COLUMNS**. Look for a star almost at the right edge. Tap A again.
3. The third says **80 COLUMNS**. Find **SUPPLIES LEFT** and the number below it. Tap A once more. **BACK AT THE HELM** puts you back in 40 columns.

The 32-column screen will look familiar to someone who used an early CoCo. The CoCo 3 adds 40- and 80-column text screens. BASIC changes how many character cells fit across the picture. Dragging the corner of the Mac window scales the picture you already have. Only one of those actions gives your program a column 48 to write in.

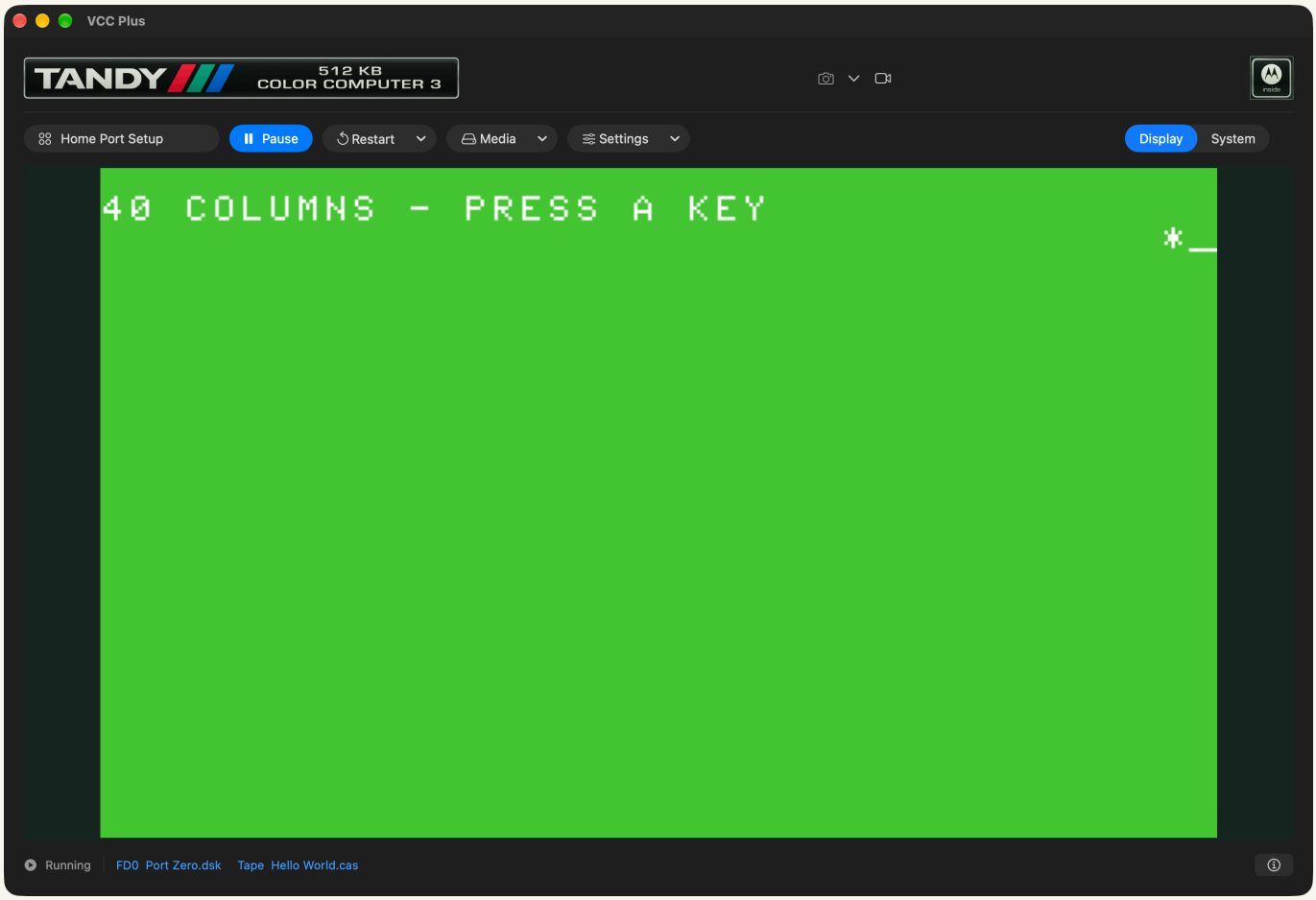


Figure 3: CONSOLE's actual 40-column screen places a star one cell in from the right edge. The Mac window also shows the camera, shared capture options and video controls.

Put one star where you want it

```
30 WIDTH 40:CLS:ATTR 3,0
40 LOCATE 0,1:PRINT "40 COLUMNS — PRESS A KEY";
50 LOCATE 38,2:PRINT "*";
60 IF INKEY$="" THEN 60
```

Think of a sheet of graph paper. CoCo 3 text coordinates count columns from the left and rows from the top. The first cell is column zero, row zero. In a 40-column row, the last column is 39. That puts the star at column 38 **one cell in from the right**. Row 2 is the third row down.

Read line 30 left to right. Select 40 columns, clear the screen, then choose the foreground and background attributes for the text to come. Line 40 moves the cursor and prints the instruction. The semicolon after the quoted text leaves the cursor on the same row. The colon between two instructions lets BASIC do them in order on a single numbered line.

Why does the star stay put? At line 60 the program checks for a keystroke over and over. It can spend a long time on one line without drawing another star. You decide when it moves on.

Make the cargo count easy to find

```
70 WIDTH 80:CLS
80 LOCATE 0,1:PRINT "80 COLUMNS";
90 LOCATE 48,1:PRINT "SUPPLIES LEFT";
100 LOCATE 0,2:PRINT "PRESS A KEY";
110 LOCATE 48,2:PRINT USING "##";6
120 IF INKEY$="" THEN 120
130 ATTR 0,1:WIDTH 40:CLS:PRINT "BACK AT THE HELM"
```

Follow the two instructions that start at column 48. The heading lands there on row 1. The number lands underneath on row 2. `PRINT USING` supplies a little printing pattern: the two number signs set aside two character places for the number six. The unused place at the left gives the digit some breathing room. Labels and numbers can line up even when a name on the left has a different length.

The actual flight reader, LOGBOOK.bas, uses the same two columns. It gets its values from a file. We will make that file in Keep a log past bedtime. For now CONSOLE gives you a rehearsal with a number already aboard.

Design decision. Would you put a two-choice landing question in 80 columns? You could, but the pilot would be reading smaller letters to make the same choice. Use the 40-column deck for the question and save the extra room for the report.

Draw a planet you can find again

Chapter 8 of 12

Somewhere on the screen is Earth. You can point to it. How could a program find it again after it has drawn it? Give Earth an address on a grid and a color number you can read back.

Mark two landing sites

Choose **RGB** under **Settings → Video Signal**. Run CHART.bas. Leave it waiting for a key. Look for a blue ring labeled EARTH and a red ring labeled MARS. Each ring has a tiny dot in the center. Press A after you have found both dots.

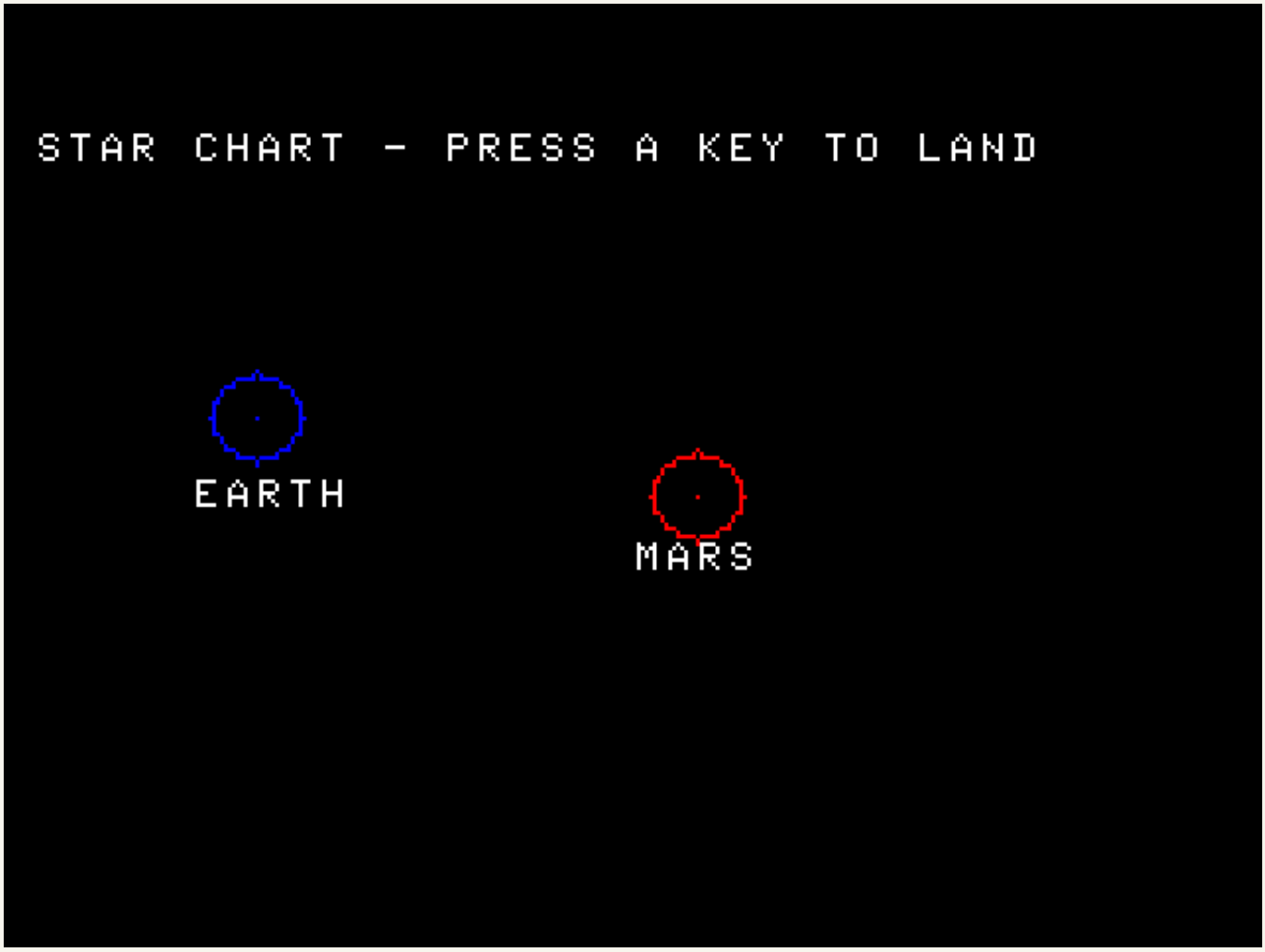


Figure 4: The guest star chart labels a blue circle Earth and a red circle Mars. Each circle has a center marker.

The program reports **EARTH MARKER 1** and **MARS MARKER 2**. Those numbers came from the dots, not the names. You could tell the planets apart from the labels even when the two colors were hard to distinguish.

```
40 HSCREEN 2:HCLS 0
42 PALETTE 0,0:PALETTE 1,9:PALETTE 2,36:PALETTE 3,63
45 HCOLOR 3:HPRINT(1,1),"STAR CHART - PRESS A KEY TO LAND"
50 FOR I=1 TO 2
60 HCIRCLE(X(I),Y(I)),12,I
```

```
70 HSET(X(I),Y(I),I)
75 HPRINT(INT(X(I)/8)-2,INT(Y(I)/8)+2),P$(I)
80 NEXT I
90 A=HPOINT(X(1),Y(1)):B=HPOINT(X(2),Y(2))
```

`HSCREEN 2` opens a CoCo 3 graphics screen. `HCLS 0` fills it with color slot zero. Think of a palette as numbered pots of paint. Line 42 puts black in pot 0, blue in 1, red in 2 and white in 3 for our RGB display. The number stored at Earth's center is **1**, even though the RGB color chosen for pot 1 is **9**. This distinction explains the report at the end.

`HCIRCLE` draws the rim. `HSET` colors the pixel at the center. `HPOINT` asks the screen which pot that pixel uses. The coordinate pairs came from the chart data. `HPRINT` places whole letters on its own grid: line 75 divides each pixel coordinate by eight, drops the fraction, and shifts the label underneath the ring. A letter covers a little block of pixels.

Keep a second picture ready

Earlier BASIC programs could draw on a coarse **SET** grid or use **PMODE** graphics. A PMODE program can keep more than one picture in RAM. Book 2's stripe experiment shows what one of those pictures looks like through composite and RGB. Here we will change which picture the monitor sees.

Run PAGES.bas in **Disk BASIC**. It first selects the 32-column screen so the following legacy graphics commands start in the intended display mode. Wait for the square at the left. Tap A once and find a smaller square on the right. Tap it again and the first square returns. A third tap brings you back to 40-column text.

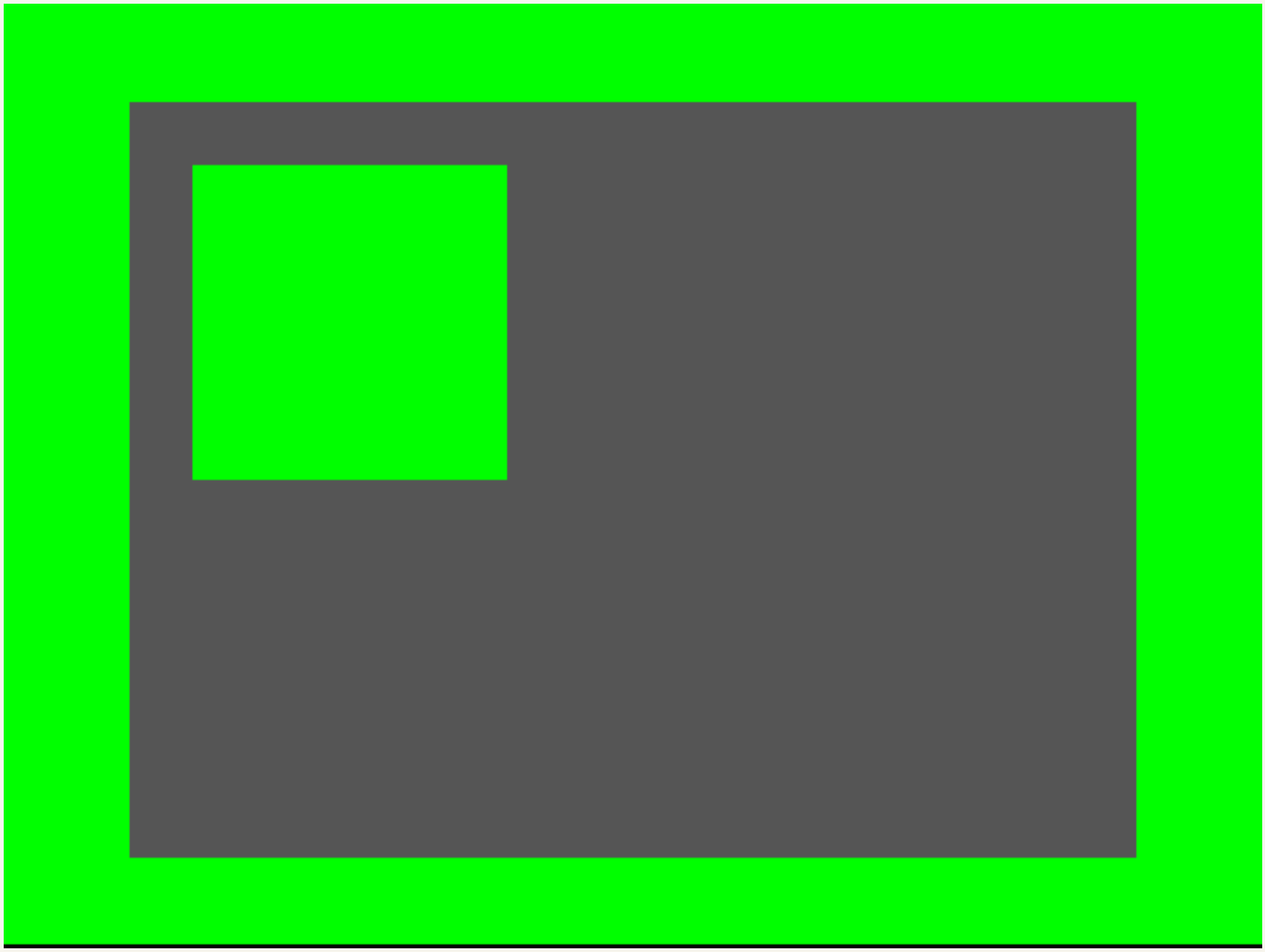


Figure 5: The first PMODE picture shows a filled square at the left. The other picture has already been drawn in memory.

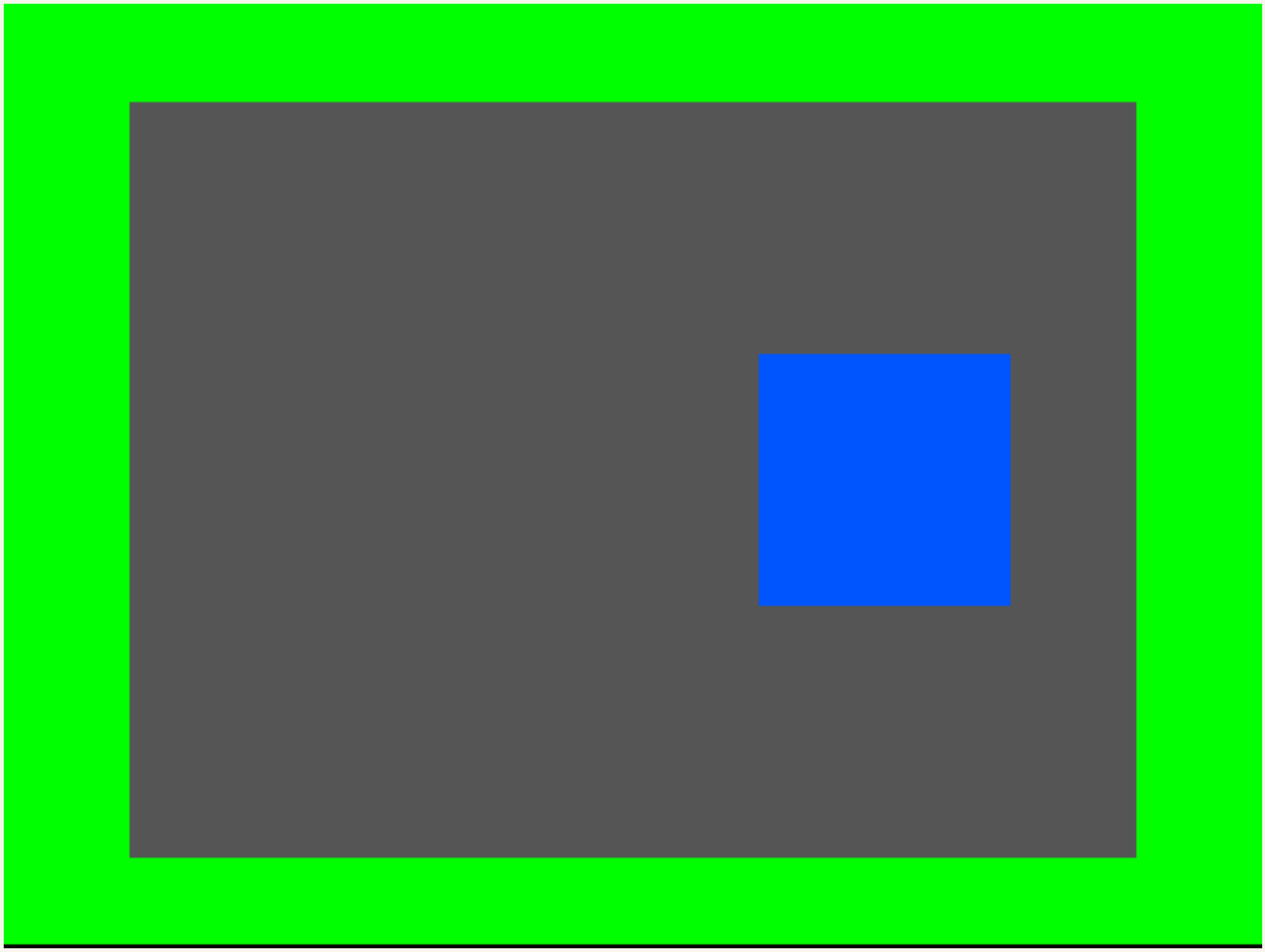


Figure 6: After the first keypress a smaller square appears on the right. The previous square is still on the other page.

```
5 WIDTH 32:CLS
10 PMODE 1,1:PCLS 0:SCREEN 1,0
20 COLOR 1,0:LINE(16,16)-(95,95),PSET,BF
30 PMODE 1,3:PCLS 0:COLOR 3,0
40 LINE(160,64)-(223,127),PSET,BF
50 IF INKEY$="" THEN 50
60 SCREEN 1,0
70 IF INKEY$="" THEN 70
80 PMODE 1,1:SCREEN 1,0
90 IF INKEY$="" THEN 90
```

Look closely at lines 30 and 40. BASIC has switched its **drawing** to another part of RAM before that first keypress. The monitor keeps showing the first square while the second is being drawn. Line 60 is the handoff: `SCREEN` shows the page BASIC has selected. Line 80 selects and shows the first one again, without drawing its square a second time. This is one way an animation can have its next picture ready in time.

Look behind the visible picture

For a closer look, put a BASIC breakpoint on line **50** of PAGES and run it again. At the stop, leave the CoCo paused. Open the **Debugger** and choose **Graphics Preview**.

1. Choose **Current Video Page**. The left square is on display.

2. Choose **At Address**, **Mapped CPU**, enter **1A00** in the hexadecimal address field and choose **Go**. Select **PMODE 1**, **Set 0**. The square on the right is already there.
3. In **Pixel Inspector**, look at X **81**, Y **32**. The byte is **$AA** at **$1E14**. You can leave its value alone. This is one byte helping hold the second picture before anyone can see it on the CoCo monitor.

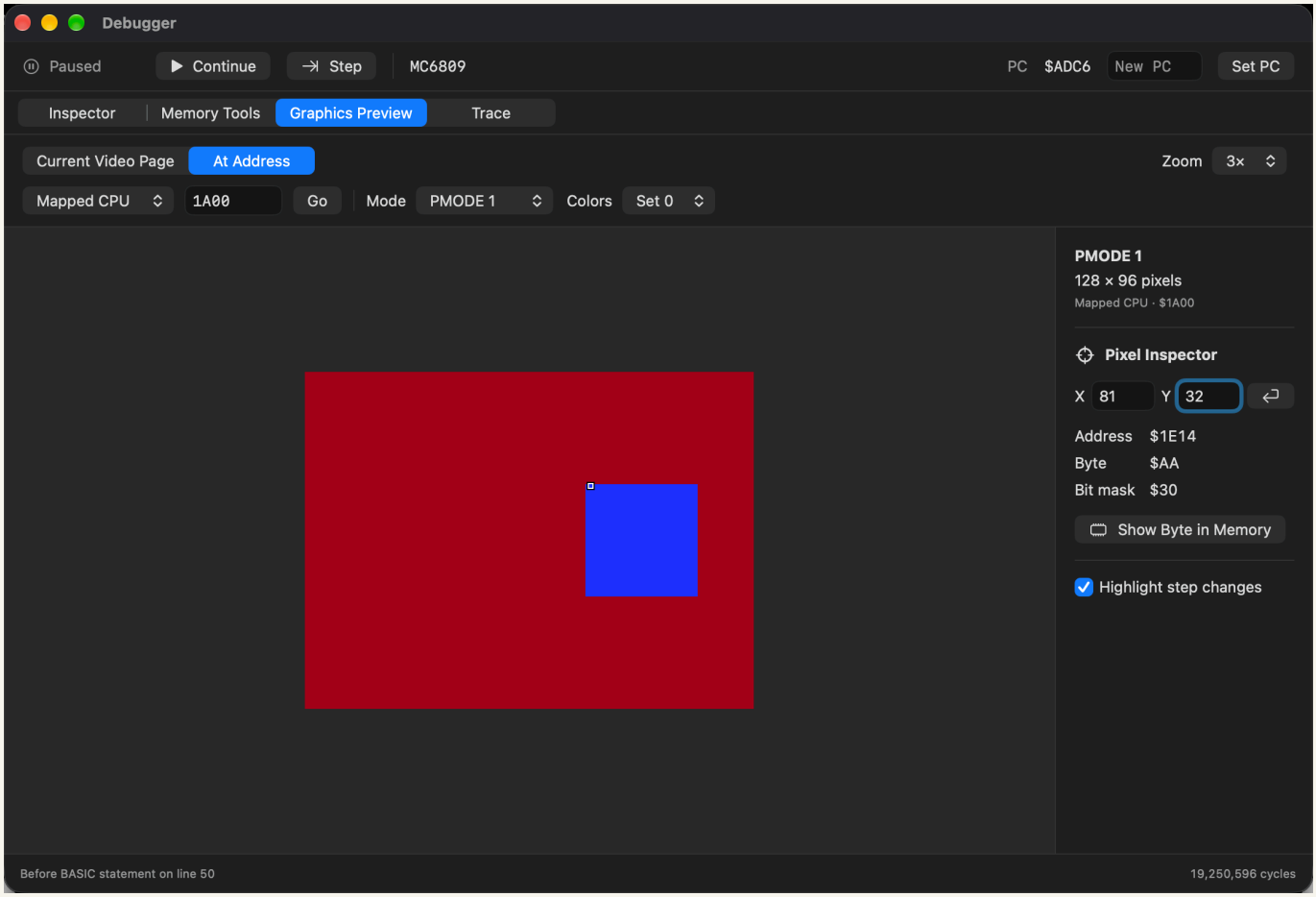


Figure 7: The native Graphics Preview inspects the right-hand square at mapped address 1A00. Pixel 81,32 uses two bits of the byte at 1E14.

Why page starts 1 and 3? One PMODE 1 picture needs two of BASIC's graphics allocation pages. The first uses pages 1 and 2; the second starts at 3. The address above applies to the Disk BASIC setup in this exercise. The preview shows its 128 by 96 stored pixels. BASIC's drawing coordinates cover a finer grid, so the preview pixel numbers differ from the coordinates in the LINE statement.

Remove the line 50 breakpoint in BASIC Development, choose **Continue**, and use the three keypresses to finish the trip. You have seen two jobs that a single block of memory can do in succession: hold a drawing under construction, then supply the image on the monitor.

Listen for a signal

Chapter 9 of 12

A joystick can report more than “left” or “right.” It can tell your program how far the stick has moved. Give the ship three messages to send, and see how one number decides which one you receive.

Point the right-port scanner

Open **Input → Keyboard and Joysticks…**. Make a note of your current **right CoCo port** assignment so you can put it back. Either select your controller for that port, or select **Keyboard** and the arrow-key layout. If the options are folded, expand the right port’s options. With the keyboard layout, Left and Right move the virtual stick and releasing the keys centers it.

Open RADAR.bas and click **Run**. Focus the CoCo display. Hold the stick toward the left, or hold Left on your keyboard, and tap **A** on the CoCo keyboard. You should get **WEST SIGNAL**. Start another run, release the direction and tap A for **CENTER SIGNAL**. On a third run, hold right and tap A for **EAST SIGNAL**. A is an ordinary typing key in the arrow layout. Choosing the W/A/S/D layout would give that key a joystick job of its own.

```
10 WIDTH 40:CLS
20 PRINT "RIGHT STICK SCANNER"
30 PRINT "MOVE THE STICK. PRESS A KEY TO SCAN."
40 IF INKEY$="" THEN 40
50 X=JOYSTK(0)
60 IF X<24 THEN PRINT "WEST SIGNAL":END
70 IF X>39 THEN PRINT "EAST SIGNAL":END
80 PRINT "CENTER SIGNAL"
```

The pause on line 40 gives you time to aim. `INKEY$` returns an empty string while there is no character waiting. When it receives a character, execution reaches line 50. `JOYSTK(0)` then samples the right port’s horizontal axis. The 0 inside the parentheses selects the axis. The number assigned to X describes its position.

Decide where west begins

The stick reports a value from 0 through 63. The middle is around 31. The program gives the center a generous landing strip.

X reading	Line that decides	Message
0 through 23	60	WEST SIGNAL
24 through 39	80	CENTER SIGNAL
40 through 63	70	EAST SIGNAL

Prediction. Suppose X is exactly 39. Line 60 asks whether 39 is smaller than 24. Line 70 asks whether it is larger than 39. Both answers are false, so BASIC reaches line 80. At 40, line 70 prints its message and executes END. The center message never gets its turn.

If all three runs return CENTER, check the right-port assignment and the selected stick. If the prompt keeps asking for a key, click the CoCo display and tap A again. Restore your previous port assignment when you finish.

Give the pilot an arrival sound

The flight in VOYAGE.bas plays three notes after the landing key and before filing the log.

```
140 PLAY "T180O3L8CEG"
150 GOSUB 200
160 PRINT "LOG SAVED FOR ";P$(D)
```

Read the short score inside the quotes. T sets the tempo to 180; O picks octave 3; L8 makes the following notes eighth notes. C, E and G are the three pitches in order. PLAY follows the score just as BASIC follows a sequence of program lines. SOUND, another command in the reference, takes a pitch number and a duration for a single tone.

During the final flight, listen when you press the landing key. The words **LOG SAVED FOR MARS** or **LOG SAVED FOR EARTH** also mark the arrival. If the Mac is muted, those words remain your check that the ship landed and its log was closed.

Keep a log past bedtime

Chapter 10 of 12

The pilot knows where the ship landed. So does the variable in RAM. Tomorrow's pilot gets a freshly started computer. How will that pilot find out where the ship went?

Give the ship its own disk

Use the **Disk BASIC** profile for this lesson. Save anything you want to keep before changing profiles or replacing the program in RAM. Open **Media → Floppy Disks…** and put a disposable, initialized floppy image in **FD0**. You can use **New Disk** in the drive actions or duplicate an initialized practice image in Finder. Keep the original somewhere else. Name the Mac working copy so that you will recognize it tomorrow.

Check the FD0 row. You want your working filename, write protection off, and **Saves automatically**. If the disk exists only in memory, use **Save As…** to give it a Mac file. If the machine is paused after the change, resume it. Back at the CoCo shows how to tell a Mac image from the CoCo files inside it.

What you hold	Where it lives	What it contains
VOYAGE.bas	A Mac source file	Instructions you can read and edit
Your working .dsk	A file in Finder, mounted in FD0	A whole emulated floppy, with room for many files
VOYAGE.DAT	A CoCo file inside that floppy	The destination and remaining supplies from the latest flight

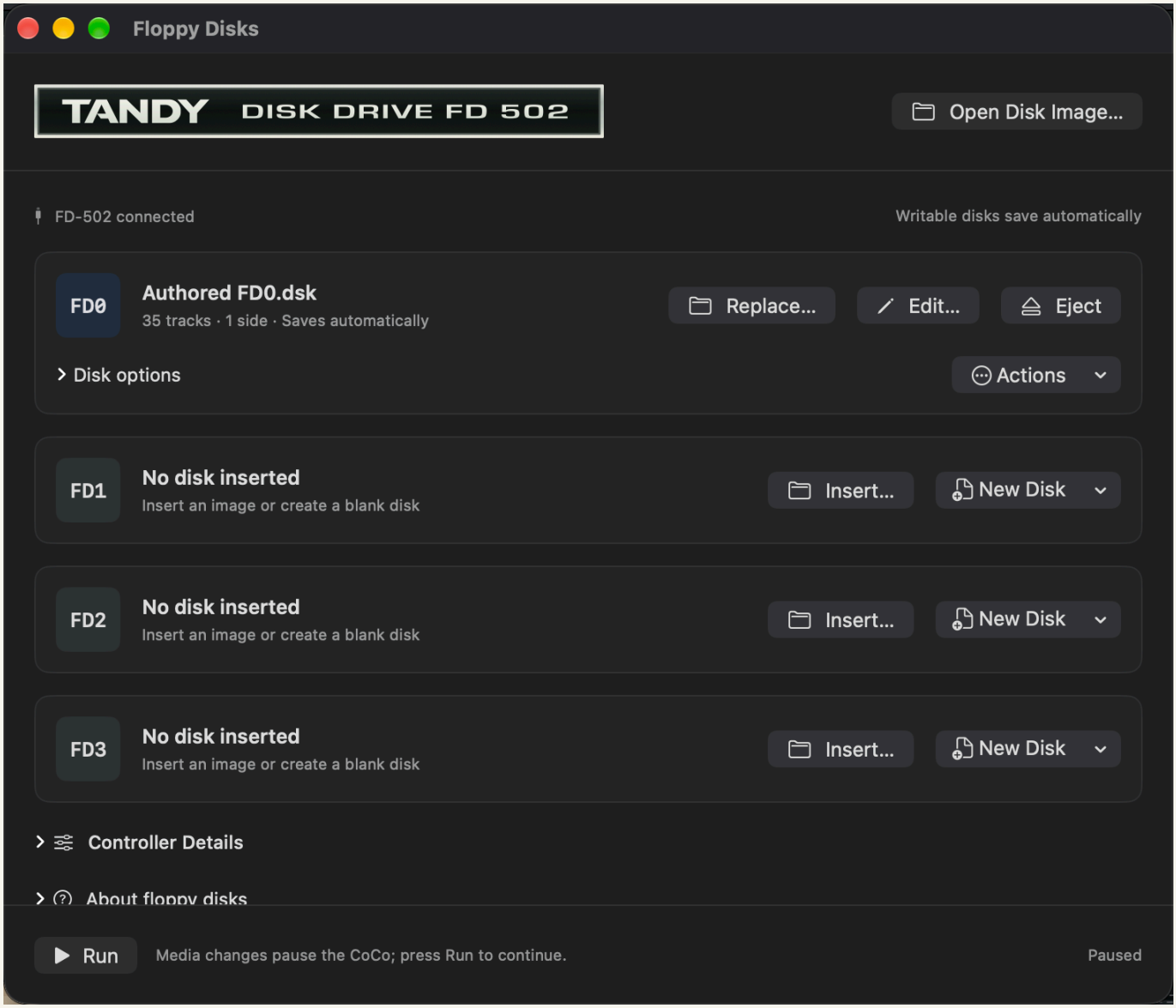


Figure 8: The current FD0 row identifies the mounted image and its save mode. This representative authored disk says Saves automatically; your own window should name the working voyage disk you chose.

Write a voyage

Open VOYAGE.bas in BASIC Development and run it. Click the CoCo display. Enter **1** and press Return for Earth. When the chart says **PRESS A KEY TO LAND**, tap A. Wait for **LOG SAVED FOR EARTH** and the BASIC prompt. Here is what happened after the landing jingle.

```
200 OPEN "O",#1,"VOYAGE.DAT:0"
210 WRITE #1,P$(D),S
220 CLOSE #1
230 RETURN
```

Line 200 opens a door between BASIC and a file. The O stands for output. **VOYAGE.DAT** is the name on the CoCo floppy, and **:0** selects drive zero. **#1** is a file number that this program can use while that door is open. The file number and drive number have different jobs.

Line 210 writes two items in order: a destination string and the number of supplies. WRITE includes the separators that another BASIC program will need to read the values. Line 220 closes the file and lets Disk BASIC finish its bookkeeping. Line 230 returns to the caller. Only after all that does the ship print its success message. Opening this file for output again replaces the previous voyage; keep this experiment on its own disk.

Hand the log to a second program

Leave FD0 in place. Open LOGBOOK.bas in BASIC Development and run it. Where did the word EARTH come from? Look at the entire reader. You will find no planet names in its source.

```
10 WIDTH 80:CLS
20 OPEN "I",#1,"VOYAGE.DAT:0"
30 INPUT #1,P$,S
40 CLOSE #1
50 LOCATE 0,1:PRINT "FLIGHT LOG";
60 LOCATE 48,1:PRINT "SUPPLIES LEFT";
70 LOCATE 0,2:PRINT P$;
80 LOCATE 48,2:PRINT S
90 END
```

The I on line 20 stands for input. Line 30 asks for the same two items in the same order in which the writer stored them. The dollar sign on P$ tells BASIC to receive a string. S receives a number. Then the reader closes the file and lays out its report. On this run the result is **EARTH** with **4** supplies left.

Try a missing log. On a different, empty initialized disk, LOGBOOK reports **?NE ERROR IN 20**. It reached the command to open the input file, but that disk has no file by the requested name. Put your voyage disk back in FD0 before running the reader again. If the BASIC workspace stopped at the error, choose **Continue** to return to the prompt first. Keep the disk with the good log; you do not need to create a blank VOYAGE.DAT to fix this.

Open locker two before locker one

A flight log can be read from the beginning. What if your ship has numbered lockers and you need the contents of locker two right now? Try LOCKERS.bas on the same disposable disk. It creates or updates a separate file called **LOCKERS.DAT**. The output names Mars first, then Earth.

```
10 WIDTH 40:CLS
20 OPEN "D",#1,"LOCKERS.DAT:0",8
30 FIELD #1,8 AS N$
40 LSET N$="EARTH":PUT #1,1
50 LSET N$="MARS":PUT #1,2
60 GET #1,2:PRINT "LOCKER TWO: ";N$
70 GET #1,1:PRINT "LOCKER ONE: ";N$
80 CLOSE #1
```

D opens a **direct-access** file. Each record has eight character places. FIELD gives those places a name in the working buffer. LSET puts a name at the left of that space. PUT sends the buffer to a particular numbered record on disk. Line 50 writes Mars to record 2. Line 60 fetches that record immediately, and line 70 goes back to fetch record 1. Disk BASIC can do those reads in the order you request.

Imagine needing 80 lockers. The fixed record length would let the system find the start of locker 80 without reading the text in the first 79. Compare that with the tiny shared log: its two items always travel together, so sequential input is enough for the job.

Read a magazine’s address book program

Bob Devries used random-access records for names and addresses in *Mini Data Base*. Open **Help → Archive Library…**, browse **Magazines → Coco Nutz! Newsletter → 2005**, and open the January issue. The article and listing begin on PDF page 10. Enter 13 in the reader’s page field and press **Return**. This page ends *Mini Data Base* and starts Richard Kelly’s *Pill*.

Above the scan, choose **Examples on this page → Mini Data Base → Open in BASIC Editor**. You now have an editable copy of the complete numbered program from the publisher’s companion ZIP. Find line 1070. Its `FIELD` statement divides a 100-character record among the parts of an address, where LOCKERS gave all eight characters to a single name. Opening the example leaves the program in the CoCo alone. **Copy Example** in the same menu puts the listing on the Mac clipboard.

Take the voyage to another machine

Chapter 11 of 12

A friend at a BASIC prompt cannot tell the CoCo to open a Mac editor window. Give the friend a disk with a real BASIC program inside, and an instruction they can type. This chapter also makes a tape for the trip. Start in **Disk BASIC** with your work saved.

Put VOYAGE on a disk

1. In **Develop → BASIC Development...**, open VOYAGE.bas. Beside Save, open **More file actions** (the button with three dots) and choose **Export Disk...**.
2. Use **VOYAGE** for the CoCo program name. For the first disk, leave **Install DOS bootloader** clear. Save the Mac image as **VOYAGE.dsk** in a practice folder. This operation makes a disk carrying your program. It does not insert the disk for you.
3. In **Media → Floppy Disks...**, insert that new image in FD0, or replace your practice disk with it. This is a fresh *working* export. Enable writing if you want to land and keep a log on this disk. Check for **Saves automatically**. If paused, resume the machine.
4. At the CoCo prompt type each of the following on its own line. Press Return after each command.

```
DIR
LOAD"VOYAGE"
RUN
```

The directory shows VOYAGE as a BASIC program. LOAD copies it into RAM. RUN reaches **FLIGHT DECK**. Choose 2, press Return, then tap A on the landing chart. Wait for **LOG SAVED FOR MARS**. Now the exported disk has a program *and* the data file that the trip just wrote.

A shorter invitation. Export a second disk from the same source and use the same CoCo name, **VOYAGE**. This time check **Install DOS bootloader** and give the Mac image a distinct name such as **VOYAGE-BOOT.dsk**. Insert it in FD0 in Disk BASIC. At the prompt type `DOS` and press Return. The loader on this particular disk starts VOYAGE and takes you to the same choice of planets. The full route, `LOAD"VOYAGE"` followed by `RUN`, also names the program you want. A plain Disk BASIC data disk needs no startup loader.

Send the program by cassette

Return to the VOYAGE source in the editor. Choose **More file actions → Export Cassette...**, set the CoCo name to **VOYAGE**, and save the Mac file as **VOYAGE.cas**. Keep a writable **practice floppy** in FD0: the program travels on tape, but its own OPEN command still sends the flight log to a disk.

Open the new cassette image in **Media → Cassette...**, rewind to the beginning, and choose **Play**. At a running BASIC prompt, type the first command below and press Return. Wait for the program to load and BASIC to return to its prompt before typing the second.

```
CLOAD"VOYAGE"
RUN
```

The recorder advances while BASIC runs its motor. The game reaches **FLIGHT DECK** from tape. Choose Mars and land to write the log on your practice floppy. If you try again, rewind the tape first. You can open LOGBOOK from the editor to read the result from FD0. A cassette holds its entries in sequence, so this time there is no floppy-style directory step.

Know which drive answered

The first two recipes used Disk BASIC and an ordinary floppy. Keep that setup if you are following the book straight through. The other BASIC profiles can run VOYAGE, too. Use disposable copies for these optional trips and keep your current Mac source saved before switching machines.

Hard Drive BASIC. RGB-DOS normally directs a request for drive zero to a hard-disk volume. Attach a writable copy of the exported floppy in **FD0**. At the prompt enter the following lines separately.

```
DRIVE OFF
DIR 0
LOAD"VOYAGE:0"
RUN
```

DRIVE OFF exposes the floppy for this route. Make sure DIR lists VOYAGE before running it. Leave floppy routing in effect through the landing **and** any LOGBOOK read, since their data file also ends in :0. When finished, use `DRIVE ON` and then `DIR 0` to see your hard-disk volume again. Do this switch at the prompt after the file has closed.

CoCo SDC. Its SD card is a Mac folder containing disk images. Put a disposable **VOYAGE.DSK** in the root of a working card folder. In the SDC cartridge details you can show the folder in Finder or choose a different one. To land and write the flight log, the **working** folder and image need to be writable and **Protect SD card files (read-only)** must be off. Preserve the original export. At the SDC-DOS prompt enter each line separately.

```
DRIVE 0,"/VOYAGE.DSK"
DIR 0
LOAD"VOYAGE:0"
RUN
```

The slash begins at the top of the card folder. The first line mounts a whole image on SDC0. The third line asks BASIC for one program *inside* that image. Leave it mounted through the landing and LOGBOOK read. Restore your usual card, mount and protection afterward. Which drive zero? shows the controls and the starter mounts. Pack a note with the exported disk. A friend needs the intended BASIC setup, the filename *inside* the disk, the command to start it, and permission to use a working copy for the new log. Those four details tell them much more than a Finder filename alone.

Fly, land, power down, remember

Chapter 12 of 12

Today's crew gets six supplies and a question. Tomorrow's crew gets a disk. Your job is to leave enough information on that disk for someone who never saw your screen to say where the ship went.

Make the handoff

1. Start in **Disk BASIC** with a disposable, initialized writable disk in FD0. Use the Mac filename and the **Saves automatically** status to identify the disk. It is fine to use your own working VOYAGE export. Keep an original copy elsewhere.
2. Open VOYAGE.bas and run it. Click the display. Try entering X first. The crew should ask again. Choose **1**, press Return, and watch the turn counts before the Earth chart appears. Tap **A** to land. Wait for **LOG SAVED FOR EARTH**.
3. Open LOGBOOK.bas and run it. **EARTH** with **4** supplies makes a useful first receipt. On paper, predict the destination **and** count the next crew will read if you fly to Mars instead.
4. Run VOYAGE again. Enter **2** for Mars and press Return. When the landing chart appears, tap A. Wait for **LOG SAVED FOR MARS** and for the Mac disk save to finish.
5. Restart the CoCo from the emulator controls. Check FD0 afterward. If you set up a fresh machine for the second crew, mount that same working disk in FD0. The reader only needs to read this time; a protected copy works.
6. From BASIC Development, open LOGBOOK and choose its **Run** button to load that reader into the newly started machine. Compare the 80-column report with your prediction.

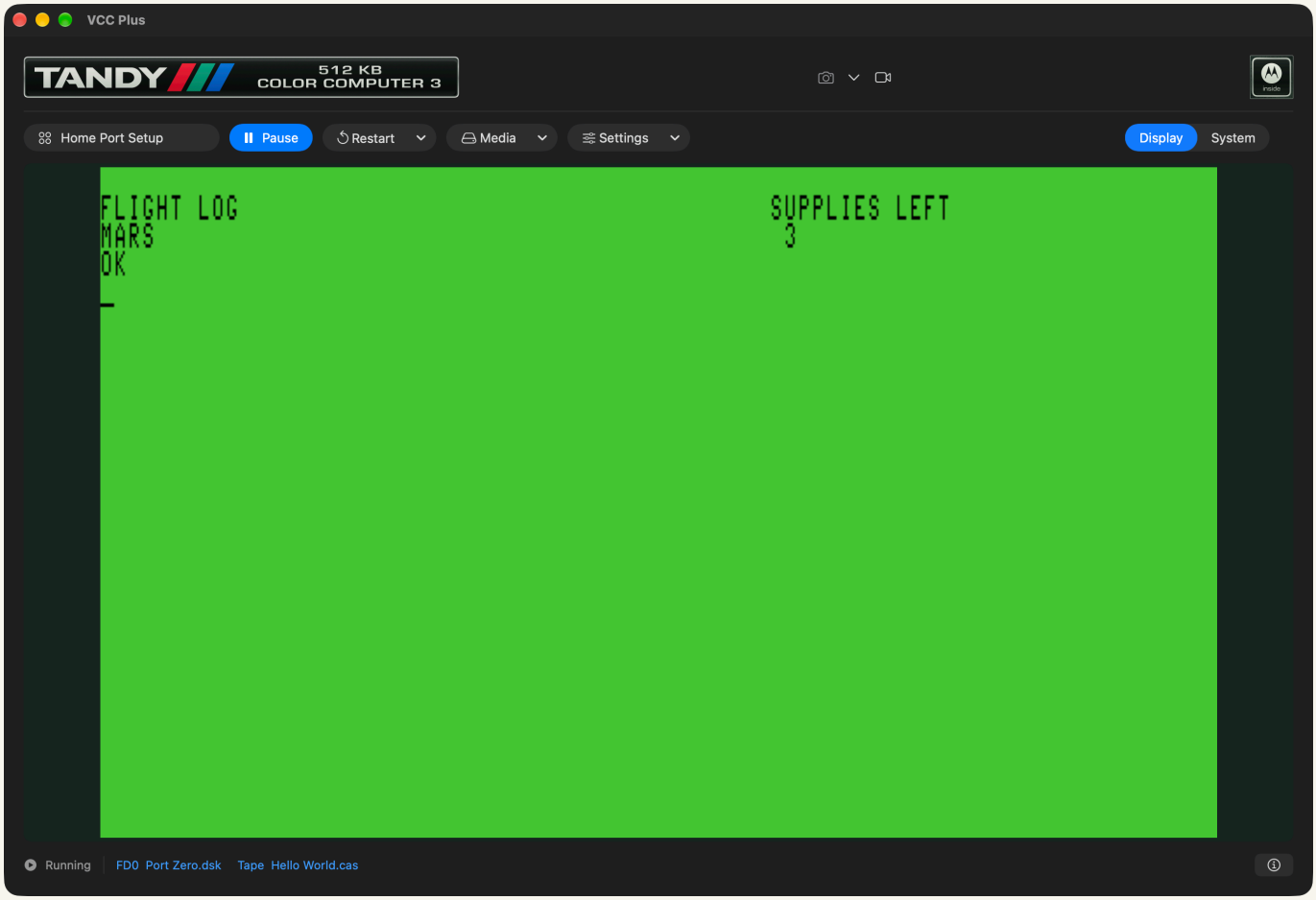


Figure 9: The flight log on the wide CoCo 3 text screen reads MARS with three supplies left.

Why three? In the flight listing, D is 1 for Earth and 2 for Mars. The loop runs through D+1 turns. Earth takes two supplies out of six. Mars takes three. The report has a destination name as well as a number, because a plain 3 could mean almost anything to a crew that missed the trip.

Reconstruct the last trip

Give the second crew the report and this set of questions. Let them try answering before opening the source.

- **Where did the pilot's keystroke go?** Line 40 of VOYAGE puts the reply in C$. Lines 50 and 60 choose a destination index. The pilot can make a mistake and get another turn.
- **What drew the landing site?** The data table connects that index to a name and two coordinates. The routine at line 300 uses them to draw and label the selected planet. Pressing a landing key releases the wait and the routine returns.
- **What survived the restart?** The machine cleared its working values. The disk file still held the name and supply count because the writer closed the file and VCC Plus retained the working image. LOGBOOK opened that file and read two values into its own variables.
- **Why is Earth missing?** The next flight opened a file of the same name for output and replaced its contents. The program promises a *last* voyage. A history of many voyages would need a different filing decision.

Recover a lost flight

What you see	First thing to check
An old planet appears on the report.	Compare the FD0 **Mac filename** with the copy used for the last landing. Check the route to drive zero if you changed BASIC profiles.
The program stops at the OPEN line with ?NE.	The reader looked for VOYAGE.DAT on the selected disk. Reattach the disk the writer used, or complete one flight on your working disk first.
The planet screen stays up.	Click the CoCo display and tap an ordinary character key. The program is waiting for permission to land.
The file never reaches the next crew's Mac image.	Check write protection, save mode and any pending save error. Let the guest close its file, then finish saving the working image before ejecting it.

For a short next project, show another person the left square in the hidden-page experiment. Ask them to guess whether the right square exists yet. You can now take them from a keypress to an instruction, from an instruction to RAM, and from RAM to the evidence on the screen. Start with whichever step makes them curious.

BASIC at your elbow

Appendix A

When you remember that `GET` can pick something up but have forgotten what goes in the parentheses, start here. The PMODE graphics form picks up a rectangle. The disk form reads a record through an open file. A program can use both.

Find a command

Browse by first letter below, or search this volume. In the app, **Read in BASIC Manual** in BASIC Quick Reference brings you straight to the entry for the selected command or error. Each name here also has its own link, so you can go straight to PRINT USING, SDC-DOS DRIVE or JOYSTK. For a message that begins with a question mark, look up its error code.

A · B · C · D · E · F · G · H · I · J · K · L · M · N · O · P · R · S · T · U · V · W

How to read a form. Uppercase words are BASIC keywords. Replace names such as `filename` or `number` with your own value. Square brackets enclose an optional part. An ellipsis means the preceding item can repeat. The forms describe how to write a statement or expression. Files, devices and variables named in a form may need setting up first. When you want a complete exercise, follow its lesson or use the named example in **Develop → BASIC Quick Reference...**.

Which machine? "CoCo 3 BASIC" includes commands provided by the extended interpreter in this machine. "Disk BASIC" needs the disk interpreter. The Hard Drive BASIC and CoCo SDC profiles add their own forms of some familiar commands. Compare the three ways to reach a disk before using a command that changes its contents.

Work out an expression

BASIC can do several jobs inside one expression. In ORDER.bas, guess both answers before running lines 20 and 30. Both use a two, a three and a four. The parentheses change which operation happens first.

```
10 WIDTH 40:CLS
20 PRINT 2+3*4
30 PRINT (2+3)*4
40 PRINT 2^3
50 PRINT 6/2-1
60 PRINT 4>3;4=3;4<>3
70 PRINT "RED"+" DUST"
80 PRINT (4>3) AND (2=2)
90 PRINT NOT (4>3)
100 END
```

The first two answers are 14 and 20. BASIC normally multiplies before adding. Parentheses let you ask for the sum first. The caret raises a number to a power. Two cubed is 8. The slash divides. Six divided by two, minus one, is 2.

The comparison signs ask questions. = asks "the same?", <> asks "different?", and <, >, <= and >= compare order. Line 60 prints **−1, 0, −1** because the answers are yes, no and yes. In an IF test, zero is false and a nonzero result is true. With text on both sides, the plus sign joins the two strings. The space at the start of the second string puts a gap in **RED DUST**.

In line 80, the two comparisons each produce a true result. AND combines them and prints −1. In line 90, NOT reverses the result of the comparison and prints 0. AND, OR and NOT can also work on a number's individual bits. If you mean to join two questions, put parentheses around each one as line 80 does.

A

ABS · AND · AS · ASC · ATN · ATTR · AUDIO

ABS

Return the magnitude of a number.

```
ABS(number)
```

CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *Numbers and rounding*.

AND

Combine integer bit patterns; often used to join conditions.

```
expression AND expression
```

CoCo 3 BASIC · Choose a destination
See also OR, NOT, IF.

AS

Give a string variable a field within a direct-access file buffer.

```
FIELD #file,length AS stringVariable
```

CoCo 3 BASIC · Keep a log past bedtime
See also FIELD.

ASC

Return the character code of the first character.

```
ASC(string)
```

The string must not be empty.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

ATN

Return the arctangent in radians.

```
ATN(number)
```

Multiply by 180/PI to convert the result to degrees.
CoCo 3 BASIC · Count the supplies

ATTR

Choose text colors and optional character attributes.

```
ATTR foreground,background[,B][,U]
```

B enables blinking; U enables underlining.
CoCo 3 high-resolution text modes.

CoCo 3 BASIC · Design a flight deck
See also WIDTH, LOCATE, PALETTE.

AUDIO

Connect or disconnect cassette playback sound from the CoCo audio output.

```
AUDIO ON
AUDIO OFF
```

Use the cassette transport and audio settings in VCC Plus to supply and hear tape playback.
CoCo 3 BASIC · Listen for a signal
See also MOTOR, CLOAD, SOUND.

B

BACKUP · BRK · BUTTON

BACKUP

Duplicate a BASIC disk onto another disk.

```
BACKUP source TO destination
BACKUP drive
```

Single-drive form requires disk exchanges.
Overwrites the destination disk. Use a new disposable image, never the only copy of your data.
Disk BASIC · Take the voyage to another machine
See also COPY.

BRK

Introduce a Break-key trap as part of ON BRK GOTO.

```
ON BRK GOTO line
```

CoCo 3 BASIC · Find the missing supply
See also ON BRK.

BUTTON

Read a joystick button on the CoCo 3.

```
BUTTON(number)
```

0/1: right first/second button; 2/3: left first/second button.
Returns 1 when pressed, otherwise 0.
CoCo 3 BASIC · Listen for a signal
See also JOYSTK.

C

CHR$ · CIRCLE · CLEAR · CLOAD · CLOADM · CLOSE · CLS · CMP · COLOR · CONT · COPY · COPY — RGB-DOS · COS · CSAVE · CSAVEM · CVN

CHR$

Return the character with the specified code.

```
CHR$(code)
```

code: 0-255.
Control codes may move the cursor or change display behavior.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

CIRCLE

Draw a circle, ellipse or arc on the PMODE screen.

```
CIRCLE(x,y),radius[,color[,ratio[,start,end]]]
```

ratio changes height relative to width.
start and end select fractions of a full turn.
CoCo 3 BASIC · Draw a planet you can find again
See also LINE, PAINT, HCIRCLE.

CLEAR

Clear variables and adjust string space or the BASIC memory ceiling.

```
CLEAR [stringBytes [, topAddress]]
```

stringBytes reserves string workspace.
topAddress leaves higher memory for machine-language code.
This discards current variable values; do not use it merely to clear the display.
CoCo 3 BASIC · Give a chart its names
See also MEM, OS — Out of string space.

CLOAD

Load a BASIC program from cassette.

```
CLOAD ["filename"]
```

Omitting the name selects the next BASIC program.
Cassette names have up to eight characters.
Loading replaces the program in CoCo memory. It does not open a Mac source document.
CoCo 3 BASIC · Take the voyage to another machine
See also CLOADM, CSAVE.

CLOADM

Load a machine-language program from cassette.

```
CLOADM ["filename"[,offset]]
```

offset relocates loaded bytes; it does not rewrite addresses inside the program.
CoCo 3 BASIC · Take the voyage to another machine
See also EXEC, CSAVEM, CLOAD.

CLOSE

Finish access to one or more open files.

```
CLOSE [#file[,#file ...]]
```

With no file number, close all open files.
Closing output files completes buffered writes.
Disk BASIC · Keep a log past bedtime
See also OPEN, UNLOAD.

CLS

Clear the text screen.

```
CLS [color]
```

color chooses a text-screen background when supplied.
CoCo 3 BASIC · Make the CoCo answer
In BASIC Quick Reference, open *Hello, CoCo*.
See also PRINT, WIDTH, HCLS, PCLS.

CMP

Restore the standard composite palette on a CoCo 3.

```
PALETTE CMP
```

CoCo 3 BASIC · Draw a planet you can find again
See also PALETTE, RGB.

COLOR

Choose foreground and background colors for PMODE graphics.

```
COLOR foreground,background
```

Available colors depend on the current graphics mode.
CoCo 3 BASIC · Draw a planet you can find again
See also PMODE, PCLS, PSET.

CONT

Continue after an ordinary BASIC stop when the interpreter still has a valid continuation.

```
CONT
```

The editor Continue button resumes its debugger stop. It does not type this command.
Changing the program can invalidate BASIC's saved continuation.
CoCo 3 BASIC · Find the missing supply
See also STOP, CN — Cannot continue.

COPY

Copy one BASIC disk file to another filename or drive.

```
COPY "source/ext:drive" TO "destination/ext:drive"
```

Include the filename extensions.
Writes to the destination disk. RGB-DOS changes overwrite behavior; see its variant entry.
Disk BASIC · Take the voyage to another machine
See also COPY — RGB-DOS, BACKUP.

COPY — RGB-DOS

Copy a file, retaining its name when the destination is only a drive number.

```
COPY "source/ext:drive" TO drive
COPY "source/ext:drive" TO "destination/ext:drive"
```

Writes the destination. Direct mode asks before replacing an existing file; inside a BASIC program, replacement occurs without that prompt.
Hard Drive BASIC (RGB-DOS) · Take the voyage to another machine
See also COPY, DRIVE — RGB-DOS.

COS

Return the cosine of an angle in radians.

```
COS(angle)
```

Angles are radians, not degrees.
CoCo 3 BASIC · Count the supplies

CSAVE

Record the current BASIC program on cassette.

```
CSAVE "filename"[,A]
```

A selects ASCII rather than tokenized program storage.
Prepare the cassette recording transport first. Export the recording in VCC Plus to retain it as a Mac file.
CoCo 3 BASIC · Take the voyage to another machine
See also CLOAD, CSAVEM.

CSAVEM

Record a memory range as a machine-language cassette file.

```
CSAVEM "filename",first,last,entry
```

first and last are inclusive byte addresses; entry is the EXEC address.
Prepare cassette recording before issuing the command.
CoCo 3 BASIC · Take the voyage to another machine
See also CLOADM, EXEC.

CVN

Decode a five-byte BASIC number stored in a string.

```
CVN(string)
```

The string must contain the expected numeric encoding.
Disk BASIC · Keep a log past bedtime
See also MKN$.

D

DATA · DEF · DEF FN · DEF USR · DEL · DIM · DIR · DIR — SDC-DOS · DLOAD · DOS · DOS — RGB-DOS · DRAW · DRIVE · DRIVE NEW · DRIVE — RGB-DOS · DRIVE — SDC-DOS · DSKI$ · DSKINI · DSKO$

DATA

Place constant values in a program for READ.

```
DATA constant [, constant ...]
```

Use quotes around strings containing commas.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Read a small data table*.
See also READ, RESTORE.

DEF

Introduce a user-defined BASIC or machine-language function.

```
DEF FNname(argument)=expression
DEF USRn=address
```

CoCo 3 BASIC · Give a routine a job
See also DEF FN, DEF USR.

DEF FN

Define a numeric function to evaluate later.

```
DEF FNname(parameter) = expression
```

FNname is invoked with a numeric argument.
CoCo 3 BASIC · Give a routine a job
In BASIC Quick Reference, open *Numbers and rounding*.
See also FN.

DEF USR

Set the entry address of a machine-language function.

```
DEF USRn = address
```

n: 0-9; the function must follow BASIC's machine-language calling convention.
CoCo 3 BASIC · Where BASIC meets the machine
See also USR, EXEC.

DEL

Delete one or more lines from the CoCo's in-memory program.

```
DEL first[–last]
DEL –last
DEL first–
```

Use care: the interpreter does not provide Mac-style Undo for this command.
CoCo 3 BASIC · Make the CoCo answer
See also LIST, NEW.

DIM

Reserve numeric or string arrays with inclusive upper bounds.

```
DIM name(upperBound [, upperBound ...]) [, ...]
```

Indices start at zero: DIM A(5) provides six elements.
Dimension an array before using it; do not dimension it twice.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Array bounds and elements*.
See also BS — Bad subscript, DD — Duplicate dimension.

DIR

List files stored on a BASIC disk.

```
DIR [drive]
```

Omit drive to list the default drive.
The listing shows name, extension, type, storage form and granule count.
In SDC-DOS, plain DIR still lists the mounted disk; DIR - browses the SD folder.
Disk BASIC · Take the voyage to another machine
In BASIC Quick Reference, open *List the mounted disk*.
See also DRIVE, DRIVE — RGB-DOS, DIR — SDC-DOS.

DIR — SDC-DOS

Browse the card folder instead of the files inside a mounted disk.

```
DIR –
DIR "path"
DIR = "path"
```

DIR = "/" selects the card root; DIR = ".." selects its parent folder.
Paths are relative to the SDC card folder selected in VCC Plus, not arbitrary Mac paths.
Plain DIR 0 still lists files inside the image mounted in drive 0.
CoCo SDC (SDC-DOS) · Take the voyage to another machine
In BASIC Quick Reference, open *Browse and mount Hello Disk*.
See also DIR, DRIVE — SDC-DOS.

DLOAD

Historical serial program-loading command.

```
DLOAD ["filename"]
```

Requires a compatible serial host and protocol.

The bundled CoCo 3 environment does not provide the historical DLOAD host service. Use native source opening or media loading instead.
CoCo 3 BASIC · Where BASIC meets the machine
See also CLOAD, LOAD.

DOS

Invoke a boot loader from drive 0.

```
DOS
```

Drive 0 must contain a compatible boot loader.
A boot loader can start another operating system or a program; DOS does not always mean OS-9.
Disk BASIC · Take the voyage to another machine
See also DOS — RGB-DOS.

DOS — RGB-DOS

Boot a selected BASIC volume, or try its AUTOEXEC program.

```
DOS [drive]
```

The NitrOS-9 preset boots automatically. Its running OS-9 shell is not a BASIC editor target.
Hard Drive BASIC (RGB-DOS) · Take the voyage to another machine
See also DOS, DRIVE — RGB-DOS.

DRAW

Draw PMODE graphics using a compact movement string.

```
DRAW string
```

U,D,L,R move in the four main directions; E,F,G,H move diagonally.
BM moves without drawing. C selects color, S scale and A angle.
X executes a string variable embedded in the drawing sequence.
CoCo 3 BASIC · Draw a planet you can find again
See also HDRAW, LINE.

DRIVE

Choose the default drive for later disk commands.

```
DRIVE number
```

Disk BASIC uses drive numbers 0-3.
RGB-DOS and SDC-DOS extend this command in different ways.
Disk BASIC · Take the voyage to another machine
See also DIR, DRIVE — RGB-DOS, DRIVE — SDC-DOS.

DRIVE NEW

Create and mount a new disk image in the selected card folder.

```
DRIVE drive,"filename.DSK",NEW
```

Writes a new file; it does not replace an existing one.

Hardware documentation also covers SDF, flash, RTC and DriveWire features. These are not supported SDC features in VCC Plus.
CoCo SDC (SDC-DOS) · Take the voyage to another machine
See also DRIVE — SDC-DOS, DIR — SDC-DOS, DSKINI.

DRIVE — RGB-DOS

Switch between virtual hard-disk volumes and floppy drives.

```
DRIVE volume
DRIVE ON
DRIVE OFF [lastFloppy]
DRIVE #unit
```

volume: 0-255 selects a default virtual BASIC disk. With DRIVE OFF, drives 0-3 refer to floppy drives instead.
DRIVE OFF exposes floppy drives; DRIVE ON restores hard-disk routing.
#unit selects the hard-disk unit. VCC Plus has HD0 and HD1.
The Hard Drive BASIC preset uses RGB-DOS. Do not confuse a virtual BASIC volume number with an HD device number.
Historical DRIVE STOP and DRIVE RESTORE hardware operations are not needed for VCC Plus image files.
Hard Drive BASIC (RGB-DOS) · Take the voyage to another machine
In BASIC Quick Reference, open *Floppy and hard-disk directories*.
See also DRIVE, DIR, DOS — RGB-DOS.

DRIVE — SDC-DOS

Inspect, mount or unload images on the SDC card.

```
DRIVE
DRIVE drive,"path"[,volume]
DRIVE drive,UNLOAD
```

drive is 0 or 1; volume selects a BASIC volume inside a compatible hard-disk image.
Use short card filenames. A mounted image is a disk containing BASIC files, not itself a BASIC program.
The same image cannot be mounted in both drives.
CoCo SDC (SDC-DOS) · Take the voyage to another machine
In BASIC Quick Reference, open *Browse and mount Hello Disk*.
See also DRIVE, DIR — SDC-DOS, DRIVE NEW.

DSKI$

Read a disk sector into two string variables.

```
DSKI$ drive,track,sector,string1,string2
```

The strings receive the two halves of a 256-byte sector.
Raw-sector access bypasses normal file interpretation.
Disk BASIC · Take the voyage to another machine
See also DSKO$.

DSKINI

Initialize a disk for BASIC file storage.

```
DSKINI [drive]
```

Select the intended drive explicitly.
Erases the selected disk and disturbs BASIC workspace. Use only a blank disposable image.
Disk BASIC · Take the voyage to another machine
See also FREE, DIR.

DSKO$

Write two strings directly to a disk sector.

```
DSKO$ drive,track,sector,string1,string2
```

Use properly sized data for the selected sector.
Raw writes can corrupt the filesystem. This is not a safe way to save a BASIC program.
Disk BASIC · Take the voyage to another machine
See also DSKI$.

E

EDIT · ELSE · END · EOF · ERLIN · ERNO · ERR · EXEC · EXP

EDIT

Use the original BASIC line editor for one program line.

```
EDIT line
```

The native BASIC editor is a separate way to edit source.
CoCo 3 BASIC · Make the CoCo answer
See also LIST, DEL, RENUM.

ELSE

Introduce the false branch of an IF statement.

```
IF condition THEN statements ELSE statements
```

CoCo 3 BASIC · Choose a destination
See also IF, THEN.

END

Finish the program.

```
END
```

CoCo 3 BASIC · Make the CoCo answer
In BASIC Quick Reference, open *Hello, CoCo*.
See also STOP, RUN.

EOF

Check whether an open input stream is at end of file.

```
EOF(file)
```

0 means more input remains; −1 indicates end of file.

Also applies to cassette input with device −1.
Disk BASIC · Keep a log past bedtime
See also OPEN, INPUT — disk file.

ERLIN

Read the BASIC line number that caused the trapped error.

```
ERLIN
```

Use inside an ON ERR handler.
CoCo 3 BASIC · Find the missing supply
In BASIC Quick Reference, open *Handle an error*.
See also ERNO, ON ERR.

ERNO

Read the numeric code for the last trapped BASIC error.

```
ERNO
```

Use inside an ON ERR handler.
CoCo 3 BASIC · Find the missing supply
In BASIC Quick Reference, open *Handle an error*.
See also ERLIN, ON ERR.

ERR

Introduce an error trap as part of ON ERR GOTO.

```
ON ERR GOTO line
```

CoCo 3 BASIC · Find the missing supply
See also ON ERR, ERNO, ERLIN.

EXEC

Transfer execution to a machine-language entry point.

```
EXEC [address]
```

Without an address, use the entry point from the last applicable machine-language load.
Running arbitrary memory as code can crash or alter the emulated machine.
CoCo 3 BASIC · Where BASIC meets the machine
See also LOADM, CLOADM, DEF USR.

EXP

Raise e to a numeric power.

```
EXP(number)
```

Large arguments can overflow.
CoCo 3 BASIC · Count the supplies

F

FIELD · FILES · FIX · FN · FOR · FREE

FIELD

Divide a direct-access buffer into string fields.

```
FIELD #file,length AS stringVariable[, ...]
```

Field lengths must fit in the opened record.
Use LSET/RSET rather than ordinary string assignment when filling fields.
Disk BASIC · Keep a log past bedtime
See also OPEN, LSET, RSET, GET — disk record, PUT — disk record.

FILES

Allocate file buffers for disk access.

```
FILES count[,size]
```

count reserves file buffers; size supplies the total buffer size.
Changes BASIC workspace allocation. Arrange file buffers before storing working data.
Disk BASIC · Keep a log past bedtime
See also OPEN, FIELD.

FIX

Remove the fractional part toward zero.

```
FIX(number)
```

FIX(-2.7) is −2.
CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *Numbers and rounding*.
See also INT.

FN

Invoke a function defined with DEF FN.

```
FNname(argument)
```

CoCo 3 BASIC · Give a routine a job
See also DEF FN.

FOR

Start a counted loop.

```
FOR variable = start TO finish [STEP increment]
```

increment defaults to 1; a negative value counts downward.
NEXT closes the loop.
CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *A counting loop*.

See also NEXT, TO, STEP.

FREE

Count free granules on a BASIC disk.

```
FREE(drive)
```

Granules are the BASIC filesystem's allocation units.
Disk BASIC · Take the voyage to another machine
See also DIR.

G

GET · GET — disk record · GO · GOSUB · GOTO

GET

Copy a PMODE rectangle into a numeric array.

```
GET(x1,y1)-(x2,y2),array[,G]
```

Dimension a two-dimensional array large enough for the rectangle.
G stores full graphics detail for PUT operations.
GET #file,record is a different form for disk files.
CoCo 3 BASIC · Draw a planet you can find again
See also PUT, GET — disk record.

GET — disk record

Read a direct-access record into its file buffer.

```
GET #file[,record]
```

Use OPEN "D" and FIELD to interpret the record.
An omitted record advances through the file.
This form is separate from graphics GET.
Disk BASIC · Keep a log past bedtime
See also FIELD, PUT — disk record, GET.

GO

Prefix of GO TO and GO SUB; the joined forms are also accepted.

```
GO TO line
GO SUB line
```

CoCo 3 BASIC · Choose a destination
See also GOTO, GOSUB.

GOSUB

Call a numbered subroutine and remember where to return.

```
GOSUB line
```

End the subroutine with RETURN.
Use the editor's Step Into, Step Over or Step Out to follow calls.
CoCo 3 BASIC · Give a routine a job
In BASIC Quick Reference, open *A returning subroutine*.
See also RETURN, ON.

GOTO

Continue execution at another numbered line.

```
GOTO line
```

line must exist in the loaded program.
CoCo 3 BASIC · Choose a destination
In BASIC Quick Reference, open *Choose a branch*.
See also GOSUB, ON, UL — Undefined line.

H

HBUFF · HCIRCLE · HCLS · HCOLOR · HDRAW · HEX$ · HGET · HLINE · HPAINT · HPOINT · HPRINT · HPUT · HRESET · HSCREEN · HSET · HSTAT

HBUFF

Reserve a buffer for high-resolution GET/PUT graphics.

```
HBUFF buffer,size
```

size is the number of bytes to reserve; maximum 7931.
CoCo 3 BASIC · Draw a planet you can find again
See also HGET, HPUT.

HCIRCLE

Draw a high-resolution circle, ellipse or arc.

```
HCIRCLE(x,y),radius[,color[,ratio[,start,end]]]
```

Optional ratio changes the shape; start/end restrict the arc.
CoCo 3 BASIC · Draw a planet you can find again
See also HLINE, HPAINT, CIRCLE.

HCLS

Fill the CoCo 3 high-resolution graphics screen.

```
HCLS [color]
```

Omitting color uses the current background.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *CoCo 3 graphics*.
See also HSCREEN, HCOLOR.

HCOLOR

Choose foreground and background palette entries for high-resolution graphics.

```
HCOLOR foreground,background
```

Use palette entries available in the selected HSCREEN mode.
CoCo 3 BASIC · Draw a planet you can find again
See also HSCREEN, PALETTE.

HDRAW

Use a movement string to draw high-resolution graphics.

```
HDRAW string
```

Uses the movement commands described under DRAW.
CoCo 3 BASIC · Draw a planet you can find again
See also DRAW, HLINE.

HEX$

Convert a number to hexadecimal text.

```
HEX$(number)
```

The returned string has no &H prefix.
CoCo 3 BASIC · Give a chart its names

HGET

Save a high-resolution rectangle in an HBUFF buffer.

```
HGET(x1,y1)-(x2,y2),buffer
```

Reserve sufficient space with HBUFF before copying.
CoCo 3 BASIC · Draw a planet you can find again
See also HBUFF, HPUT, GET.

HLINE

Draw a high-resolution line or rectangle.

```
HLINE [(x1,y1)]-(x2,y2),color[,B or BF]
```

B draws a box; BF fills it.
color may be a palette entry, PSET or PRESET.
CoCo 3 BASIC · Draw a planet you can find again
See also HSET, HCIRCLE, LINE.

HPAINT

Fill a bounded high-resolution region.

```
HPAINT(x,y)[,color[,border]]
```

Select a point inside a closed border.
CoCo 3 BASIC · Draw a planet you can find again
See also HCIRCLE, HLINE, PAINT.

HPOINT

Read the color of a high-resolution graphics point.

```
HPOINT(x,y)
```

Coordinates follow the selected HSCREEN mode.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *CoCo 3 graphics*.
See also HSET.

HPRINT

Draw text in a CoCo 3 high-resolution graphics display.

```
HPRINT(column,row),string
```

Coordinates select character cells, not individual pixels.
CoCo 3 BASIC · Draw a planet you can find again
See also HSCREEN, HCOLOR, PRINT.

HPUT

Restore a high-resolution rectangle from a buffer.

```
HPUT(x1,y1)-(x2,y2),buffer,action
```

action: PSET, PRESET, AND, OR or NOT.
CoCo 3 BASIC · Draw a planet you can find again
See also HBUFF, HGET, PUT.

HRESET

Reset a high-resolution point to the background.

```
HRESET(x,y)
```

CoCo 3 BASIC · Draw a planet you can find again
See also HSET, HPOINT.

HSCREEN

Select a CoCo 3 high-resolution graphics display.

```
HSCREEN mode
```

0: return to low-resolution display.
1: 320x192, 4 colors; 2: 320x192, 16 colors.
3: 640x192, 2 colors; 4: 640x192, 4 colors.
Selecting modes 1-4 also clears that graphics display.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *CoCo 3 graphics*.
See also HCLS, HCOLOR, PALETTE.

HSET

Set a point in CoCo 3 high-resolution graphics.

```
HSET(x,y[,color])
```

Omitting color uses the foreground.
Coordinates depend on the HSCREEN resolution.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *CoCo 3 graphics*.
See also HRESET, HPOINT.

HSTAT

Read the high-resolution text character, attributes and cursor coordinates.

```
HSTAT character,attribute,column,row
```

Supply numeric variables to receive the four results.
CoCo 3 BASIC · Design a flight deck
See also LOCATE, ATTR, WIDTH.

I

IF · INKEY$ · INPUT · INPUT — disk file · INSTR · INT

IF

Choose statements or a destination line according to a test.

```
IF condition THEN statements [ELSE statements]
IF condition THEN line [ELSE line]
```

Zero is false; a nonzero value is true.
A false condition skips the remainder of its branch on that BASIC line.
CoCo 3 BASIC · Choose a destination
In BASIC Quick Reference, open *Choose a branch*.
See also THEN, ELSE, AND, OR.

INKEY$

Read a waiting key without stopping for input.

```
INKEY$
```

The result is an empty string when no key is waiting.
CoCo 3 BASIC · Listen for a signal

INPUT

Read keyboard values into numeric or string variables.

```
INPUT ["prompt";] variable [, variable ...]
```

Separate multiple responses with commas.
Use LINE INPUT when a whole line, including commas, belongs in one string.
CoCo 3 BASIC · Choose a destination
See also LINE INPUT, INKEY$.

INPUT — disk file

Read delimited values from an open input file.

```
INPUT #file,variable[,variable ...]
```

Variable types must match the stored fields.
Disk BASIC · Keep a log past bedtime
See also OPEN, WRITE, EOF.

INSTR

Find the first occurrence of a substring.

```
INSTR([start,]string,target)
```

Positions start at 1; a zero result means no match.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

INT

Round down to an integer, including for negative values.

```
INT(number)
```

INT(-2.7) is −3.
CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *Numbers and rounding*.
See also FIX.

J

JOYSTK

JOYSTK

Read an emulated joystick axis.

```
JOYSTK(axis)
```

0: right X; 1: right Y; 2: left X; 3: left Y.
Values range from 0 to 63; center is about 31.
CoCo 3 BASIC · Listen for a signal
See also BUTTON.

K

KILL

KILL

Remove a file from a BASIC disk directory.

```
KILL "filename/ext[:drive]"
```

Include the extension.
Deletes the file from the selected disk; the command has no Undo.
Disk BASIC · Take the voyage to another machine
See also DIR, RENAME.

L

LEFT$ · LEN · LET · LINE · LINE INPUT · LINE INPUT — disk file · LIST · LLIST · LOAD · LOADM · LOC · LOCATE · LOF · LOG · LPEEK · LPOKE · LSET

LEFT$

Take characters from the beginning of a string.

```
LEFT$(string,count)
```

count is the maximum number of characters returned.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

LEN

Count the characters in a string.

```
LEN(string)
```

CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

LET

Assign a value to a variable; LET is optional.

```
[LET] variable = expression
```

Use a name ending in $ for a string.
Only the first two characters of a variable name distinguish it in this BASIC.
Changing source in the editor does not change variables in a running CoCo program.
CoCo 3 BASIC · Choose a destination
In BASIC Quick Reference, open *Numbers and rounding*.
See also DIM.

LINE

Draw a PMODE line, rectangle or filled rectangle.

```
LINE [(x1,y1)]-(x2,y2),action[,B or BF]
```

action is PSET or PRESET; B draws a rectangle and BF fills it.
If the first point is omitted, drawing starts at the previous endpoint.
B or BF can be used with either PSET or PRESET.
CoCo 3 BASIC · Draw a planet you can find again
See also PSET, CIRCLE, HLINE.

LINE INPUT

Read an entire line into a string variable.

```
LINE INPUT ["prompt";] stringVariable
```

Commas in the entered line remain part of the string.
CoCo 3 BASIC · Choose a destination
See also INPUT, LINE INPUT — disk file.

LINE INPUT — disk file

Read an entire file line into one string.

```
LINE INPUT #file,stringVariable
```

Commas remain part of the string.
Disk BASIC · Keep a log past bedtime
See also OPEN, EOF, LINE INPUT.

LIST

Display a program or a range of BASIC lines.

```
LIST [first[-last]]
LIST -last
LIST first-
```

A single line number lists only that line.
Read from CoCo brings program text into the native editor without typing LIST.
CoCo 3 BASIC · Make the CoCo answer
See also LLIST, EDIT.

LLIST

Send a program listing to the printer device.

```
LLIST [first[-last]]
```

Enable VCC Plus Printer Output capture to collect printer text.
CoCo 3 BASIC · Design a flight deck
See also LIST, PRINT.

LOAD

Load a BASIC program from disk.

```
LOAD "filename"[,R]
```

R runs the program after loading.
A :drive suffix selects a drive instead of the default.
Loading replaces the BASIC program in CoCo memory.
Disk BASIC · Take the voyage to another machine
See also SAVE, RUN — disk file.

LOADM

Load a machine-language file from disk.

```
LOADM "filename"[,offset]
```

offset changes where bytes are loaded; internal machine-code addresses are not relocated.
Use EXEC after loading unless the file implements its own autostart.
Disk BASIC · Take the voyage to another machine
See also EXEC, SAVEM.

LOC

Read the current record number for an open disk file.

```
LOC(file)
```

file refers to an open disk buffer.
Disk BASIC · Keep a log past bedtime
See also OPEN, LOF.

LOCATE

Position the cursor in the CoCo 3 high-resolution text screen.

```
LOCATE column,row
```

Coordinates start at zero.
Use WIDTH 40 or WIDTH 80 first.
CoCo 3 BASIC · Design a flight deck
See also WIDTH, ATTR, HSTAT.

LOF

Read the highest record number in an open disk file.

```
LOF(file)
```

This is a record count, not a Mac file-size query.
Disk BASIC · Keep a log past bedtime
See also OPEN, LOC.

LOG

Return the natural logarithm.

```
LOG(number)
```

number must be positive.
CoCo 3 BASIC · Count the supplies

LPEEK

Read a byte using the CoCo 3 long-memory interface.

```
LPEEK(address)
```

address: 0-524287. This BASIC interface is limited to 512 KB.
CoCo 3 BASIC · Where BASIC meets the machine
See also LPOKE.

LPOKE

Write a byte using the CoCo 3 BASIC long-memory interface.

```
LPOKE address,value
```

address: 0-524287; value: 0-255.
This command's address range does not expand simply because the emulator has more RAM.
CoCo 3 BASIC · Where BASIC meets the machine
See also LPEEK, POKE.

LSET

Left-align text in a direct-access field.

```
LSET stringField = string
```

Short strings are padded; long strings are limited to the field length.
Disk BASIC · Keep a log past bedtime
See also FIELD, RSET.

M

MEM · MERGE · MID$ · MKN$ · MOTOR

MEM

Report currently free BASIC memory.

```
MEM
```

Available BASIC workspace is not the same as the selected total machine RAM.
CoCo 3 BASIC · Where BASIC meets the machine

MERGE

Combine a disk ASCII listing with the program in memory.

```
MERGE "filename"[,R]
```

Save the source file with SAVE ...,A first.
R starts the merged program.
Duplicate line numbers are replaced by the incoming lines.
Disk BASIC · Take the voyage to another machine
See also LOAD, SAVE.

MID$

Read a substring, or replace characters in a string variable.

```
MID$(string,start[,count])
MID$(stringVariable,start[,count]) = replacement
```

Positions start at 1.
Without count, the function reads to the end.
Assignment changes existing characters; it does not insert new space.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

MKN$

Encode a BASIC number into its five-byte representation.

```
MKN$(number)
```

Use with fixed-length binary fields; this is not decimal text.
Disk BASIC · Keep a log past bedtime
See also CVN, FIELD.

MOTOR

Control the cassette motor signal.

```
MOTOR ON
MOTOR OFF
```

CoCo 3 BASIC · Take the voyage to another machine
See also AUDIO, CLOAD, CSAVE.

N

NEW · NEXT · NOT

NEW

Erase the program currently held by BASIC.

```
NEW
```

This changes CoCo memory, not a saved Mac source file.
CoCo 3 BASIC · Make the CoCo answer
See also CLEAR, RUN.

NEXT

Advance a FOR loop and test whether it should repeat.

```
NEXT [variable [, variable ...]]
```

For nested loops, close the innermost loop first.
CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *A counting loop*.
See also FOR, NF — NEXT without FOR.

NOT

Invert an integer bit pattern or a BASIC truth value.

```
NOT expression
```

CoCo 3 BASIC · Choose a destination
See also AND, OR, IF.

O

OFF · ON · ON BRK · ON ERR · OPEN · OR

OFF

Disable a feature in commands that accept an ON/OFF switch.

```
AUDIO OFF
MOTOR OFF
VERIFY OFF
```

CoCo 3 BASIC · Listen for a signal
See also AUDIO, MOTOR, VERIFY.

ON

Select a GOTO or GOSUB destination by its position in a list.

```
ON expression GOTO line [, line ...]
ON expression GOSUB line [, line ...]
```

A value of 1 selects the first destination.
Zero or a value beyond the list falls through.
CoCo 3 BASIC · Give a routine a job
In BASIC Quick Reference, open *Choose a branch*.
See also GOTO, GOSUB, ON ERR, ON BRK.

ON BRK

Branch to a handler when the CoCo Break key is pressed.

```
ON BRK GOTO line
```

CoCo 3 extension.
ON BRK GOTO 0 restores ordinary Break handling.
CoCo 3 BASIC · Find the missing supply
See also STOP, ON ERR.

ON ERR

Send runtime errors to a BASIC handler on the CoCo 3.

```
ON ERR GOTO line
```

Use ERNO and ERLIN in the handler to identify the error.
ON ERR GOTO 0 disables the handler.

Handled errors remain under BASIC control; VCC Plus does not replace them with its unhandled-error pause.
CoCo 3 BASIC · Find the missing supply
In BASIC Quick Reference, open *Handle an error*.
See also ERNO, ERLIN, SN — Syntax error.

OPEN

Open a data stream to a file or device.

```
OPEN "mode",#file,"filename"[,recordLength]
```

I reads sequential data; O writes sequential data; D selects direct-access disk records.
Disk file numbers: 1-15, subject to FILES allocation.
Device −1 is cassette; −2 is printer; 0 is console.
O can replace an existing file; use a disposable output filename while learning.
Disk BASIC · Keep a log past bedtime
See also CLOSE, FILES, FIELD.

OR

Combine conditions or integer bit patterns when either side is set.

```
expression OR expression
```

CoCo 3 BASIC · Choose a destination
See also AND, NOT, IF.

P

PAINT · PALETTE · PCLEAR · PCLS · PCOPY · PEEK · PLAY · PMODE · POINT · POKE · POS · PPOINT · PRESET · PRINT · PRINT USING · PRINT — file or printer · PRINT@ · PSET · PUT · PUT — disk record

PAINT

Fill a bounded PMODE area from a starting point.

```
PAINT(x,y)[,color[,border]]
```

The starting point must be inside a closed border for a bounded fill.
CoCo 3 BASIC · Draw a planet you can find again
See also CIRCLE, LINE, HPAINT.

PALETTE

Assign a CoCo 3 palette entry or restore a standard palette.

```
PALETTE register,color
PALETTE CMP
PALETTE RGB
```

register: 0-15; color: 0-63.
CMP and RGB choose standard tables for the corresponding monitor signal.
The BASIC palette and VCC Plus Video Signal setting are related but separate controls.
CoCo 3 BASIC · Draw a planet you can find again
See also HSCREEN, HCOLOR, CMP, RGB.

PCLEAR

Reserve 1.5 KB pages for PMODE graphics.

```
PCLEAR pages
```

Reserve enough pages for the chosen PMODE and starting page.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *Draw and inspect a point*.
See also PMODE, PCOPY, MEM.

PCLS

Fill the current PMODE graphics screen.

```
PCLS [color]
```

Omitting color uses the current background.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *Draw and inspect a point*.
See also PMODE, COLOR, CLS.

PCOPY

Copy one reserved PMODE graphics page to another.

```
PCOPY source TO destination
```

Pages must have been reserved with PCLEAR.
CoCo 3 BASIC · Draw a planet you can find again
See also PCLEAR, PMODE.

PEEK

Read a byte through the current CPU memory map.

```
PEEK(address)
```

address: 0-65535.
Reading device registers can affect the device.
CoCo 3 BASIC · Where BASIC meets the machine
See also POKE.

PLAY

Play notes described by a music string.

```
PLAY string
```

A-G name notes; O selects octave, L note length and T tempo.
or + sharpens a note; - flattens it; P inserts a pause.
CoCo 3 BASIC · Listen for a signal
See also SOUND.

PMODE

Choose the resolution and starting page for traditional CoCo graphics.

```
PMODE mode,page
```

0: 128x96, 2 colors; 1: 128x96, 4 colors.
2: 128x192, 2 colors; 3: 128x192, 4 colors.
4: 256x192, 2 colors.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *Draw and inspect a point*.
See also PCLEAR, SCREEN, PCLS.

POINT

Read a semigraphics block on the text screen.

```
POINT(x,y)
```

-1 indicates part of an ordinary text character; otherwise the result describes the block color.
CoCo 3 BASIC · Draw a planet you can find again
See also SET.

POKE

Write one byte through the current CPU memory map.

```
POKE address,value
```

address: 0-65535; value: 0-255.
Writes can change programs, device registers or system state. Use only addresses you understand.
CoCo 3 BASIC · Where BASIC meets the machine
See also PEEK, LPOKE.

POS

Read the current print-column position for a device.

```
POS(device)
```

device 0 is the screen; −2 is the printer.
CoCo 3 BASIC · Design a flight deck

PPOINT

Read the color of a PMODE graphics point.

```
PPOINT(x,y)
```

CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *Draw and inspect a point*.
See also PSET.

PRESET

Reset a PMODE point to the background color.

```
PRESET(x,y)
```

CoCo 3 BASIC · Draw a planet you can find again
See also PSET, PPOINT.

PRINT

Display values on the text screen.

```
PRINT [expression [; or , expression] ...]
```

Semicolon joins output without a print-zone advance.
Comma advances to the next print zone.
A trailing separator suppresses the usual newline.
? is a shorthand for PRINT.
CoCo 3 BASIC · Make the CoCo answer
In BASIC Quick Reference, open *Hello, CoCo*.
See also PRINT@, USING, TAB.

PRINT USING

Print values with a reusable layout.

```
PRINT USING "format";expression[;expression ...]
```

Each # reserves a digit position; a decimal point places the fractional part.
For strings, ! prints one character and % followed by spaces and % defines a fixed-width field.
CoCo 3 BASIC · Design a flight deck
In BASIC Quick Reference, open *Formatted output*.
See also PRINT, USING, TAB.

PRINT — file or printer

Write formatted values to an open file or output device.

```
PRINT #file,expression[; or , expression ...]
```

-2 selects the printer; −1 selects cassette.
Positive file numbers refer to previously opened disk files.
Disk/cassette output writes media. Printer text is captured only when the VCC Plus printer is enabled.
Disk BASIC · Keep a log past bedtime
See also OPEN, CLOSE, WRITE, USING.

PRINT@

Place text at a numbered position on the 32-column screen.

```
PRINT @position, expression
```

position: 0-511, counted across rows from the top left.
CoCo 3 BASIC · Design a flight deck
See also PRINT, LOCATE, WIDTH.

PSET

Set a point in the PMODE drawing area.

```
PSET(x,y[,color])
```

Drawing coordinates use x: 0-255 and y: 0-191.
Omit color to use the current foreground.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *Draw and inspect a point*.
See also PRESET, PPOINT.

PUT

Draw a PMODE rectangle from a saved numeric array.

```
PUT(x1,y1)-(x2,y2),array[,action]
```

action: PSET, PRESET, AND, OR or NOT.
Use a matching rectangle and storage prepared by GET.
CoCo 3 BASIC · Draw a planet you can find again
See also GET, PUT — disk record.

PUT — disk record

Write the file buffer as a direct-access record.

```
PUT #file[,record]
```

Use LSET/RSET to fill the buffer before writing.
This changes the opened disk file. It is separate from graphics PUT.
Disk BASIC · Keep a log past bedtime
See also FIELD, GET — disk record, PUT.

R

READ · REM · RENAME · RENUM · RESET · RESTORE · RETURN · RGB · RIGHT$ · RND · RSET · RUN · RUN — disk file · RUNM

READ

Take the next values from the program's DATA statements.

```
READ variable [, variable ...]
```

Variable type must match the corresponding value.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Read a small data table*.
See also DATA, RESTORE, OD — Out of DATA.

REM

Add a comment that BASIC does not execute.

```
REM comment
```

An apostrophe also starts a comment. Quoted strings and comments retain their entered case.
CoCo 3 BASIC · Make the CoCo answer

See also DATA.

RENAME

Change a filename within the same BASIC disk.

```
RENAME "old/ext[:drive]" TO "new/ext[:drive]"
```

Include both extensions.
Changes the disk directory.
Disk BASIC · Take the voyage to another machine
See also DIR, COPY.

RENUM

Renumber program lines and update recognized line references.

```
RENUM [newStart [, oldStart [, increment]]]
```

newStart is the first assigned number.
oldStart selects the first old line to change.
increment controls spacing.
The editor Renumber action supports Undo and updates editor source rather than the current CoCo program.
CoCo 3 BASIC · Find the missing supply
See also LIST.

RESET

Clear a semigraphics block on the text screen.

```
RESET(x,y)
```

x: 0-63; y: 0-31.
CoCo 3 BASIC · Draw a planet you can find again
See also SET, POINT.

RESTORE

Move the DATA pointer back to the first item.

```
RESTORE
```

CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Read a small data table*.
See also READ, DATA.

RETURN

Resume immediately after the matching GOSUB call.

```
RETURN
```

RETURN needs an active GOSUB call.
CoCo 3 BASIC · Give a routine a job
In BASIC Quick Reference, open *A returning subroutine*.
See also GOSUB, RG — RETURN without GOSUB.

RGB

Restore the standard RGB palette on a CoCo 3.

```
PALETTE RGB
```

CoCo 3 BASIC · Draw a planet you can find again
See also PALETTE, CMP.

RIGHT$

Take characters from the end of a string.

```
RIGHT$(string,count)
```

count is the maximum number of characters returned.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

RND

Generate a pseudo-random value.

```
RND(limit)
```

RND(0) returns a fractional value.
A positive integer limit returns an integer from 1 through limit.
CoCo 3 BASIC · Count the supplies

RSET

Right-align text in a direct-access field.

```
RSET stringField = string
```

Short strings are padded on the left.
Disk BASIC · Keep a log past bedtime
See also FIELD, LSET.

RUN

Start the program in memory, normally clearing its variables.

```
RUN [line]
```

Omit line to start at the beginning.
Use the editor's Run button to transfer edited source and start it.
CoCo 3 BASIC · Make the CoCo answer
In BASIC Quick Reference, open *Hello, CoCo*.
See also END, RUN — disk file.

RUN — disk file

Load and run a BASIC program from disk.

```
RUN "filename"[,R]
```

R preserves open files while chaining to the new program.
Replaces the in-memory BASIC program.
Disk BASIC · Take the voyage to another machine
See also LOAD, RUN.

RUNM

Load and execute a machine-language program in one command.

```
RUNM "filename[:drive]"
```

For binaries that start themselves during loading, use LOADM instead; RUNM can execute their entry point again.
Hard Drive BASIC (RGB-DOS) · Take the voyage to another machine
See also LOADM, EXEC.

S

SAVE · SAVEM · SCREEN · SET · SGN · SIN · SKIPF · SOUND · SQR · STEP · STOP · STR$ · STRING$ · SUB

SAVE

Write the in-memory BASIC program to disk.

```
SAVE "filename"[,A]
```

A selects an ASCII listing instead of tokenized storage.
Writes to the selected image and may replace an existing file. The native editor Save command instead saves a Mac source file.
Disk BASIC · Take the voyage to another machine
See also LOAD, CSAVE.

SAVEM

Write a memory range as a machine-language disk file.

```
SAVEM "filename",first,last,entry
```

first/last are inclusive addresses; entry is the default EXEC address.
Writes to the selected disk image.
Disk BASIC · Take the voyage to another machine
See also LOADM, EXEC.

SCREEN

Switch between the traditional text and graphics displays.

```
SCREEN type,colorSet
```

type: 0 for text, 1 for PMODE graphics.
colorSet: 0 or 1.
CoCo 3 BASIC · Draw a planet you can find again
In BASIC Quick Reference, open *Draw and inspect a point*.
See also PMODE, HSCREEN.

SET

Color one semigraphics block on the 32-column text screen.

```
SET(x,y,color)
```

x: 0-63; y: 0-31; color: 0-8.
CoCo 3 BASIC · Draw a planet you can find again
See also RESET, POINT, PSET.

SGN

Return −1, 0 or 1 for a negative, zero or positive number.

```
SGN(number)
```

CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *Numbers and rounding*.

SIN

Return the sine of an angle in radians.

```
SIN(angle)
```

A full turn is 2*PI radians; use a numeric value for PI.
CoCo 3 BASIC · Count the supplies

SKIPF

Advance through a cassette file without loading it.

```
SKIPF ["filename"]
```

CoCo 3 BASIC · Take the voyage to another machine
See also CLOAD, CLOADM.

SOUND

Play a tone for a specified duration.

```
SOUND pitch,duration
```

pitch: 1-255; duration: 1-255.
CoCo 3 BASIC · Listen for a signal
See also PLAY, AUDIO.

SQR

Return the nonnegative square root.

```
SQR(number)
```

number must be zero or positive.
CoCo 3 BASIC · Count the supplies
In BASIC Quick Reference, open *Numbers and rounding*.

STEP

Set the increment of a FOR loop.

```
FOR variable=start TO finish STEP increment
```

CoCo 3 BASIC · Count the supplies
See also FOR, NEXT.

STOP

Stop execution and report the BASIC line.

```
STOP
```

A BASIC STOP is distinct from an editor breakpoint.
CoCo 3 BASIC · Find the missing supply
See also CONT, END.

STR$

Convert a number into a printable string.

```
STR$(number)
```

Positive numbers normally include a leading space.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

STRING$

Create a string containing a repeated character.

```
STRING$(count,codeOrString)
```

A string argument supplies its first character.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Formatted output*.

SUB

Complete the split spelling GO SUB.

```
GO SUB line
```

CoCo 3 BASIC · Give a routine a job
See also GOSUB, RETURN.

T

TAB · TAN · THEN · TIMER · TO · TROFF · TRON

TAB

Advance within a PRINT line to a column.

```
PRINT TAB(column);expression
```

CoCo 3 BASIC · Design a flight deck
See also PRINT, POS.

TAN

Return the tangent of an angle in radians.

```
TAN(angle)
```

Angles near odd multiples of PI/2 produce very large results.
CoCo 3 BASIC · Count the supplies

THEN

Introduce the true branch of an IF statement.

```
IF condition THEN statements
```

CoCo 3 BASIC · Choose a destination
See also IF, ELSE.

TIMER

Read or set the interpreter's ticking counter.

```
TIMER
TIMER = value
```

The counter wraps at 65536.
It advances with the emulated machine, not a guaranteed wall-clock duration.
CoCo 3 BASIC · Listen for a signal

TO

Separate start/end values or source/destination operands.

```
FOR variable=start TO finish
```

CoCo 3 BASIC · Count the supplies
See also FOR, COPY, BACKUP.

TROFF

Turn off the BASIC line-number trace.

```
TROFF
```

CoCo 3 BASIC · Find the missing supply
See also TRON.

TRON

Show BASIC line numbers as the program executes.

```
TRON
```

This is the interpreter's text trace; it is independent of editor breakpoints.
CoCo 3 BASIC · Find the missing supply
See also TROFF.

U

UNLOAD · USING · USR

UNLOAD

Close open files associated with a disk.

```
UNLOAD [drive]
```

Omitting drive selects the default drive.
This BASIC command does not eject the Mac disk-image file from VCC Plus.
Disk BASIC · Take the voyage to another machine
See also CLOSE, DRIVE — SDC-DOS.

USING

Apply a numeric or string layout to printed values.

```
PRINT USING "format";expression
```

CoCo 3 BASIC · Design a flight deck
In BASIC Quick Reference, open *Formatted output*.
See also PRINT, PRINT — file or printer.

USR

Call a machine-language function defined with DEF USR.

```
USRn(argument)
```

n: 0-9. Define a valid entry address and calling convention first.
CoCo 3 BASIC · Where BASIC meets the machine
See also DEF USR.

V

VAL · VARPTR · VERIFY

VAL

Convert a numeric prefix of a string into a number.

```
VAL(string)
```

Parsing follows BASIC numeric notation.
CoCo 3 BASIC · Give a chart its names
In BASIC Quick Reference, open *Working with strings*.

VARPTR

Find the address of a variable's internal storage.

```
VARPTR(variable)
```

BASIC may move storage later; do not assume the address remains valid.
CoCo 3 BASIC · Where BASIC meets the machine

VERIFY

Choose whether disk writes are followed by a verification read.

```
VERIFY ON
VERIFY OFF
```

Verification checks the data written to the emulated disk.
This is an interpreter setting, not the image's write-protection setting.
Disk BASIC · Take the voyage to another machine
See also SAVE.

W

WIDTH · WRITE

WIDTH

Select the CoCo 3 text-screen width.

```
WIDTH columns
```

columns: 32, 40 or 80.
40- and 80-column modes have 24 rows.
CoCo 3 extension.
CoCo 3 BASIC · Design a flight deck
See also LOCATE, ATTR, PRINT.

WRITE

Write comma-delimited data suitable for INPUT.

```
WRITE #file,expression[,expression ...]
```

String values are quoted automatically.
Writes to the opened output file.
Disk BASIC · Keep a log past bedtime
See also OPEN, INPUT, INPUT — disk file.

Where BASIC meets the machine

Appendix B

When you ask BASIC how many supplies are left, you use the name S. Somewhere inside the CoCo, the interpreter keeps bytes for that number. In this little tour you can look at one of those bytes without changing it.

Find a value and its address

Open ADDRESS.bas. Line 20 puts the number 42 in A. The next three lines ask different questions about the same variable. Which line do you expect to print 42?

```
10 WIDTH 40:CLS
20 A=42
30 PRINT "VALUE";A
40 PRINT "ADDRESS ";HEX$(VARPTR(A))
50 PRINT "FIRST BYTE";PEEK(VARPTR(A))
60 END
```

The output labeled **VALUE** is 42. The **ADDRESS** is where BASIC put the variable's data this time. HEX$ displays the address in hexadecimal, a number system with sixteen digits per place. After 9 its digits are A, B, C, D, E and F. Your address can include letters.

The last line with PRINT asks for **one byte**, starting at that address. In the supplied exercise we saw address 2667 in hexadecimal and first byte 134 in decimal. BASIC stores this numeric variable as several bytes. Reading just the first byte gives a piece of the stored number. To learn the value of A while debugging, ask for A in Watches. The whole value is already there.

Use the address reported in **your** session when inspecting memory. Adding program lines or variables can move the data, even if A still holds 42. The VARPTR, HEX$ and PEEK entries describe the three operations in the listing.

Follow the workspace

Think of BASIC's workspace as part of RAM that the interpreter manages on your behalf. It keeps the numbered program, variables and space for strings there. MEM reports the space BASIC still has available. The RAM badge at the top of the emulator reports the capacity of the emulated machine. A program, a screen, and the interpreter can occupy different pieces of that machine.

The processor uses 16-bit addresses: it can name 65,536 byte positions in its current view. The CoCo 3 can contain much more RAM than this. Its memory management unit, or MMU, maps blocks of physical memory into the processor's view. In the debugger this is the distinction between a CPU address and a physical address. It explains why a number from VARPTR only describes the current arrangement.

PEEK reads from the current address space. POKE writes a byte there. A write can change a variable, an instruction, or a hardware setting, depending on the address. Use a separate practice session for an experiment that writes to memory. The CoCo 3's LPEEK and LPOKE accept long addresses up through 524287. That BASIC interface covers the first 512 KB, including on a machine configured with more RAM.

Hand control to machine code

BASIC lines are stored in a form that the interpreter reads. A machine-language program contains instructions that the processor executes. Each has its own loading path. A disk BASIC program such as the one in the export lesson can be loaded by name and started with RUN. LOADM brings machine-code blocks from disk into memory. EXEC transfers control to an entry address. The assembler and the program's loading instructions tell you the right address.

A machine routine can also do one job for a BASIC program. DEF USR assigns a routine's entry address to a USR slot. USR calls the routine with an argument. The caller and routine have to agree on how data goes in and how the result comes back. For an ordinary BASIC subroutine like the landing routine, GOSUB and RETURN already handle the trip.

To explore CPU instructions on the Mac, open **Develop → Assembly Quick Reference…**. For loading old programs, remember that DLOAD once expected a compatible host on a serial connection. VCC Plus provides native source opening and emulated media loading; select the one appropriate to the file you have.

Historical sources and credits

- Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17; functions, pp. 18–22; operators, p. 23; error codes, p. 27. The short format of the lookup entries follows the same practical idea: a reader at the keyboard should be able to find one operation quickly.
- InterTAN Canada Ltd. *Color Computer Disk BASIC Quick Reference Guide*. 1987. Disk commands, pp. 3–8; disk errors, p. 9. Useful for comparing sequential files with direct-access records.
- Krupski, Roger, Joe Scinta, Greg Miller, and Colin J. Stearman. *RGB-DOS*. Ken-Ton Electronics. COPY, DRIVE, DOS and RUNM, PDF pp. 6–8. Page references here use the pages of the scan.
- Lindner, Tim. *CoCo SDC User Guide*. 4th ed., February 2023. Based on the original manual by Brian Blake; CoCo SDC hardware and software by Darren Atkinson. Drive status and mounting, pp. 9–10; directory and startup operations, pp. 11–12.

The Color Computer Archive preserves historical documentation that informed this book. The lessons and programs here were written for VCC Plus. The printed sources describe period equipment or DOS extensions; the flight exercises, screen captures and Mac controls show how to work with this emulator.

What did that error mean?

Appendix C

A short message can tell you exactly where to start looking. On our empty practice disk, the flight-log reader stopped with **?NE ERROR IN 20**. The two letters identify the problem. The number after IN is the line BASIC was trying to execute.

Look up the code

/0 · AE · AO · BR · BS · CN · DD · DF · DN · DS · ER · FC · FD · FM · FN · FO · FS · HP · HR · ID · IE · IO · LS · NE · NF · NO · OB · OD · OM · OS · OV · RG · SE · SN · ST · TM · UL · VF · WP

For the flight-log example, NE means “file not found.” Open LOGBOOK.bas and look at line 20. It asks drive zero to open a file called VOYAGE.DAT for input. Is the practice disk in FD0? Has the crew already landed and saved a voyage? Either answer is more helpful than changing line 20 at random.

For a program of your own, note the code and line number before starting over. Choose that line in the editor and read it together with the line above it. A bad value can be assigned much earlier than the line that first tries to use it. A breakpoint and a watch let you catch the change while the program is still running.

Check the value

Give each operation the sort of value it expects.

/0 · FC · LS · OV · OS · ST · TM

/0 — Division by zero

A denominator evaluated to zero. Check the values at the reported line.

See also IF, Choose a destination.

FC — Illegal function call

An argument is outside the range accepted by this command or function. Check its parameter descriptions.

See also SQR, ASC, Choose a destination.

LS — String too long

The result exceeds BASIC's maximum string length of 255 characters.

See also LEN, LEFT$, MID$, Choose a destination.

OV — Overflow

A numeric result is too large for BASIC to represent.

See also EXP, Choose a destination.

OS — Out of string space

The reserved string workspace is exhausted. Consider reserving more with CLEAR before creating data.

See also CLEAR, LEN, Choose a destination.

ST — String expression too complex

Simplify the expression by saving intermediate string results in variables.

See also LET, MID$, Choose a destination.

TM — Type mismatch

An expression uses a string where a number is required, or the reverse.

See also VAL, STR$, INPUT, Choose a destination.

Trace the program

Start at the reported line. Check how control reached it.
CN · DS · ID · NF · RG · SN · UL

CN — Cannot continue

BASIC no longer has a valid continuation after a stop. Restart with RUN if appropriate.
See also CONT, RUN, STOP, Find the missing supply.

DS — Direct statement in file

A program being loaded contains a statement without a BASIC line number.
See also LOAD, MERGE, Find the missing supply.

ID — Illegal direct statement

This statement requires execution from a numbered program rather than the immediate prompt.
See also RUN, Find the missing supply.

NF — NEXT without FOR

Execution reached NEXT without a matching active loop. Check loop entry paths.
See also FOR, NEXT, Find the missing supply.

RG — RETURN without GOSUB

There is no active subroutine call to return from. Avoid falling into a subroutine from the main program.
See also GOSUB, RETURN, END, Find the missing supply.

SN — Syntax error

BASIC could not interpret a statement. Check spelling, punctuation, quotes and the command form.
See also PRINT, Find the missing supply.

UL — Undefined line

A branch or run command refers to a line that is not in the program.
See also GOTO, GOSUB, RUN, Find the missing supply.

Count the items

Compare the number of items you set aside with the number you used.
BS · DD · OD · OM

BS — Bad subscript

An array index is outside its declared bounds or has the wrong number of dimensions.
See also DIM, Give a chart its names.

DD — Duplicate dimension

An array has already been dimensioned or implicitly created. Dimension it before its first use.
See also DIM, Give a chart its names.

OD — Out of DATA

READ requested more values than the DATA statements provide.
See also DATA, READ, RESTORE, Give a chart its names.

OM — Out of memory

BASIC has insufficient workspace for the operation. Check arrays, strings, graphics pages and program size.
See also MEM, CLEAR, DIM, PCLEAR, Give a chart its names.

Check the screen mode

Match the graphics mode, coordinates and buffers to the drawing command.
HP · HR

HP — High-resolution print error

The requested high-resolution text operation is invalid. Check the display mode and coordinates.
See also HPRINT, HSCREEN, Draw a planet you can find again.

HR — High-resolution graphics error

A CoCo 3 graphics argument or buffer is invalid. Check coordinates, mode and allocated buffer size.
See also HSCREEN, HBUFF, HGET, HPUT, Draw a planet you can find again.

Check the open file

Look for OPEN, its file number and its mode, then follow each read or write.
AO · BR · ER · FD · FM · FO · IE · NO · OB · SE

AO — Already open

The file is already open. Close it before reopening; on SDC, an image cannot be mounted twice.
See also CLOSE, OPEN, DRIVE — SDC-DOS, Keep a log past bedtime.

BR — Bad record

The direct-access record number is invalid.
See also GET — disk record, PUT — disk record, LOC, LOF, Keep a log past bedtime.

ER — End of record

A direct-access operation went beyond the current record boundary. Check record size and fields.
See also OPEN, FIELD, Keep a log past bedtime.

FD — Bad file data

The next input value cannot be read into the requested variable type.
See also INPUT, INPUT — disk file, WRITE, Keep a log past bedtime.

FM — File mode

The operation does not match the mode used to open the file.
See also OPEN, Keep a log past bedtime.

FO — Field overflow

FIELD definitions exceed the file buffer's record size.
See also FIELD, OPEN, Keep a log past bedtime.

IE — Input past end

The program tried to read beyond the end of a file. Test EOF before requesting another item.
See also EOF, Keep a log past bedtime.

NO — File not open

Open the file before reading, writing or requesting its records.
See also OPEN, CLOSE, Keep a log past bedtime.

OB — Out of buffers

No suitable disk buffer is available. Allocate the required file buffers before opening files.
See also FILES, OPEN, Keep a log past bedtime.

SE — Not a field string

LSET or RSET was given a string that is not currently mapped by FIELD.
See also FIELD, LSET, RSET, Keep a log past bedtime.

Check the disk or device

Check the filename and the device that answered the request.
AE · DF · DN · FN · FS · IO · NE · VF · WP

AE — File already exists

The destination name is already in use. Choose another name or use the documented replacement behavior.
See also COPY, COPY — RGB-DOS, RENAME, Take the voyage to another machine.

DF — Disk full

There is insufficient space for the write. Use another working image or remove unneeded files carefully.
See also FREE, DIR, Take the voyage to another machine.

DN — Device number

The selected device or drive number is not valid for this operation.
See also OPEN, DRIVE, Take the voyage to another machine.

FN — Bad filename

The filename is not valid for the selected filesystem. Check its length, extension and drive syntax.
See also LOAD, SAVE, Take the voyage to another machine.

FS — Bad file structure

The disk file or filesystem structure is inconsistent with the requested operation.
See also DIR, Take the voyage to another machine.

IO — Input/output error

The requested media operation failed. Check mounting, format, filename and the selected drive.
See also DIR, DRIVE, CLOAD, Take the voyage to another machine.

NE — File not found

The named disk file was not found on the selected drive. Check DIR, extension and drive suffix.
See also DIR, LOAD, DRIVE, Take the voyage to another machine.

VF — Verification error

Data read back after a write did not verify. Preserve another copy before continuing to modify the image.
See also VERIFY, Take the voyage to another machine.

WP — Write protected

The selected media is read-only. Change protection only if you intend to modify that working copy.

See also SAVE, DRIVE — SDC-DOS, Take the voyage to another machine.

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

The VCC Plus User’s Handbook

Find the key your game is waiting for, read the lights on a disk drive, or bring a favorite setup back. Start with the question you have while the CoCo is running.



VCC Plus application icon with the Color Computer and its three color bars

Brian K. Holman

Book 4 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Find your way around 3
Who has the keyboard? 15
Read and hear the CoCo 19
Connect a controller 27
Keep a setup you like 30
Change the machine 33
Find something to run 37
Put the right file in the right device 47
Find a workbench or a reference 62
Get back to a working setup 69
Look up a control 72
The hardware behind the controls 76
Publication and artwork notices 77

Find your way around

Chapter 1 of 10

You can tell quite a lot about a CoCo before pressing a single key. Look at the top of its window to see which machine you have. Look at the bottom to see whether it is running and what you have put in its drives. The part in between is yours to rearrange.

Bring the app onto your Mac

For the packaged edition, use a Mac running macOS 14 or later, with an Apple Silicon or Intel processor. Open the VCC Plus disk image in Finder. Copy **VCC Plus** to **Applications**, let the copy finish, eject the disk image, and open the copy in Applications. The disk image on the Mac delivers the app. A CoCo disk image, such as a DSK, is a file you put into one of its emulated drives.

The packaged setup used in this book includes the machine ROM, disk controller firmware and starter media. The first launch makes local working copies for you. Saving a BASIC program to your starter disk writes to your own copy. If your edition asks you to open a ROM, choose the appropriate CoCo 3 system ROM that you have available; the hardware section explains how this differs from choosing a cartridge. The development tools and the books work locally. Looking up a title in the Software Library and downloading that title are separate actions.

Read your desk

On a fresh installation with the supplied ROMs, VCC Plus starts in **Disk BASIC**. The small **Using your CoCo** card tells you how to get going with that particular setup. Click outside the card to put it away. The ⓘ button at the lower right brings it back. If someone else has already set up this Mac, look at the profile name and the attached media before changing anything.

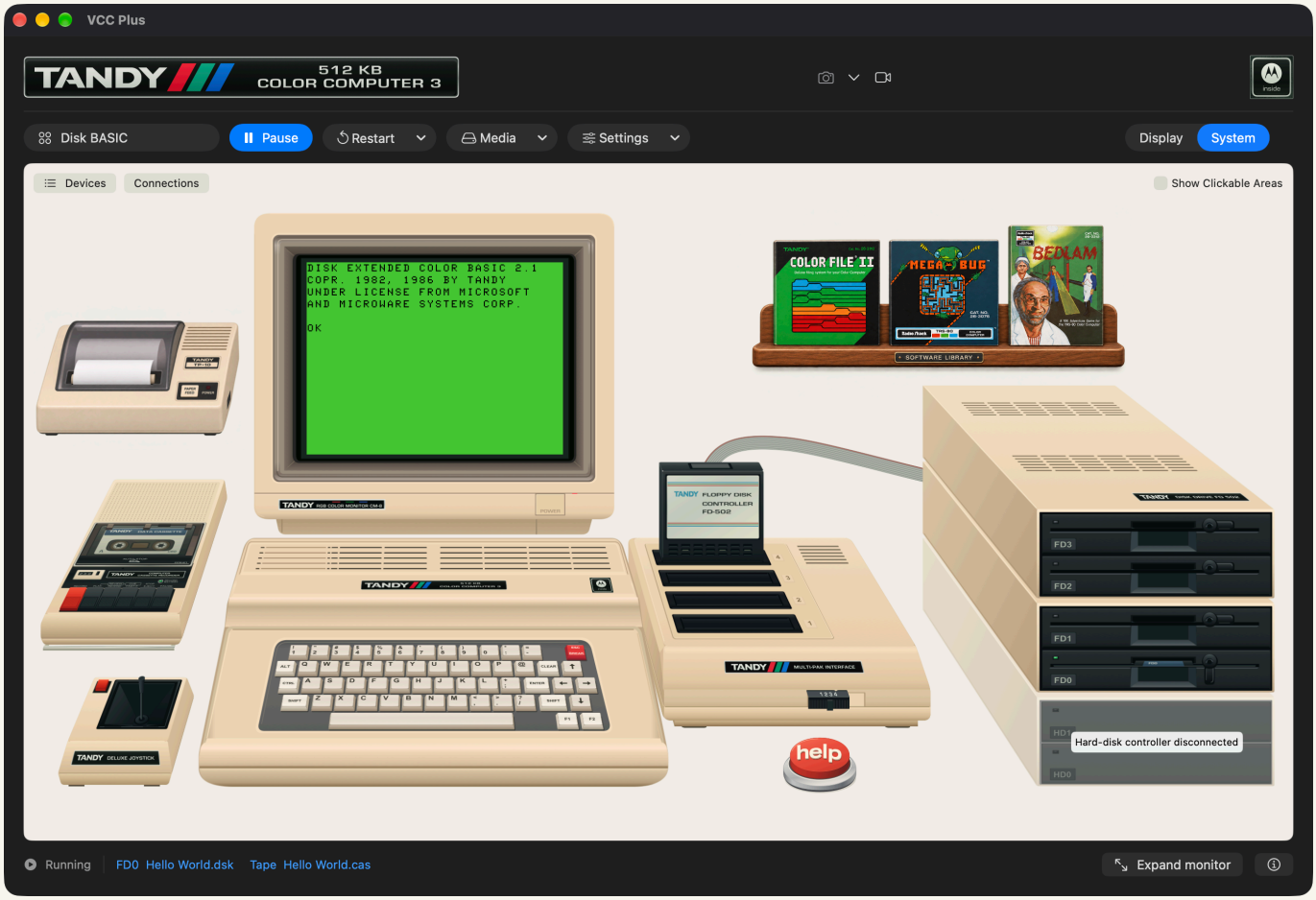


Figure 1: System view shows the running Disk BASIC setup, with Hello World.dsk in FD0 and Hello World.cas attached. Camera, capture options and video controls sit above the toolbar; the information button at the lower right opens the setup guide.

Start with three landmarks:

- **The nameplate and the badge.** The number on the left is the amount of RAM in this CoCo. The badge on the right identifies its processor. Click either to inspect Machine Settings. You can look and close the window without applying a change.
- **The screen.** The lettering inside the monitor comes from the CoCo. On a Disk BASIC startup screen, **OK** means BASIC is ready for a command. A game may draw its own title or ask you for a key instead. **Pause** in the toolbar means the machine is running and you can pause it. **Run** means you can start or resume it.
- **The bottom edge.** Read the status word at the left. A filename beside **FD0** identifies the disk image in floppy drive zero. Select that name to inspect the drive. A tape or hard-disk name opens its corresponding window. A name here tells you which medium is attached. To find a program *inside* it, use the guest's directory or loading instructions.

Locate a device without guessing

Choose **Display** → **Interactive System**. Move the pointer over the monitor's lower bezel, then over a drive face. The blue outline follows the target. **Show Clickable Areas** draws a thin dashed outline around all available targets. Turn it off when you want to see the artwork on its own.

Try one round trip. Click **Devices** at the upper left, find floppy drive zero and open its disclosure. Compare the filename there with the FD0 name below the picture. The list also reaches a Multi-Pak slot hidden behind a cartridge. **Connections** helps trace each group of drives back to its controller. Turn it on when you want to know how a disk request reaches a drive.

On the desk	What you can do there
Wooden bookshelf	Open the Software Library to look for something to run.
Keyboard case or joystick	Find input controls. Click the **screen** when you want to type into the guest.
Tape recorder or printer	Inspect the transport or open Printer Output. Export a new tape or save the captured printout when you want to keep it.
Monitor bezel	Choose a video signal. Click **Expand monitor** below the picture for a larger live display.
Multi-Pak switch and numbered slots	Inspect the startup slot and see which cartridge or controller occupies each position. An actual slot change may restart the CoCo.
Floppy or hard-drive face	Read its name and status. Open its drive window for all file choices.

Make room for the display

Choose **Display → Expand Monitor**, or press **Shift-Command-E** from System view. The monitor comes forward inside the same window. Choose **Return to System** to put it back. Try **Display → Conventional Display** if you want the picture to occupy most of the window as you work. The **System** and **Display** buttons in the toolbar make the same choice.

The running program, sound and attached media belong to the machine. Switching views lets you look at it another way. A custom hardware combination may require conventional controls. If you see **Custom configuration**, use **Cartridges…** to inspect the actual devices. Your installed cards can stay where they are.

Keep the emulator and this book side by side

In System view, click the red **help** button below the Multi-Pak to open the manual index. From either view, **Help → Manual Library…** brings the reader forward. Move the reader to one side and the emulator to the other. From a chapter, choose a numbered Contents entry to change chapters, or a topic below the current chapter to jump within it. Use **Book contents** to search this volume, or **Manual Library** to search the whole shelf. Select a figure to open it at full size and go Back to your place.

The reader remembers where you left it. Close and reopen the window, or quit and relaunch VCC Plus, then open Manual Library to continue. Each book keeps its own reading position. A topic link, search result or bookmark takes you to that specific location rather than restoring a different page. **Library Home** still returns to the shelf deliberately. These reading choices stay on this Mac; opening a book does not restore or run a CoCo program.

Keep a manual bookmark

While reading, choose the toolbar's **Bookmark this location** icon. Its filled bookmark symbol identifies a location already saved; clicking it again finds the existing bookmark instead of adding a duplicate. You can keep several useful places in each book. Reading farther changes your last-read position without moving those bookmarks.

Choose **Bookmarks**, the list icon beside it, to open the one bookmark list. **All Books** groups locations by book; **This Book** shows only the book you are reading. Bundled CMOC and LWTools reference manuals appear under their own names. The list is for the manuals, separate from Software Library's Favorites.

Select a bookmark to open it. Its actions menu also offers **Open**, **Open Book**, **Rename…** and **Remove Bookmark**. Rename supplies your own reminder while retaining the original section name. Removing a bookmark does not remove a book or its contents. If an updated book no longer has that exact section, the reader keeps the bookmark and reports **Location changed**. Use Open Book to find a new place deliberately.

Bookmarks and reading positions stay outside disposable document caches and are not included in update-check statistics.

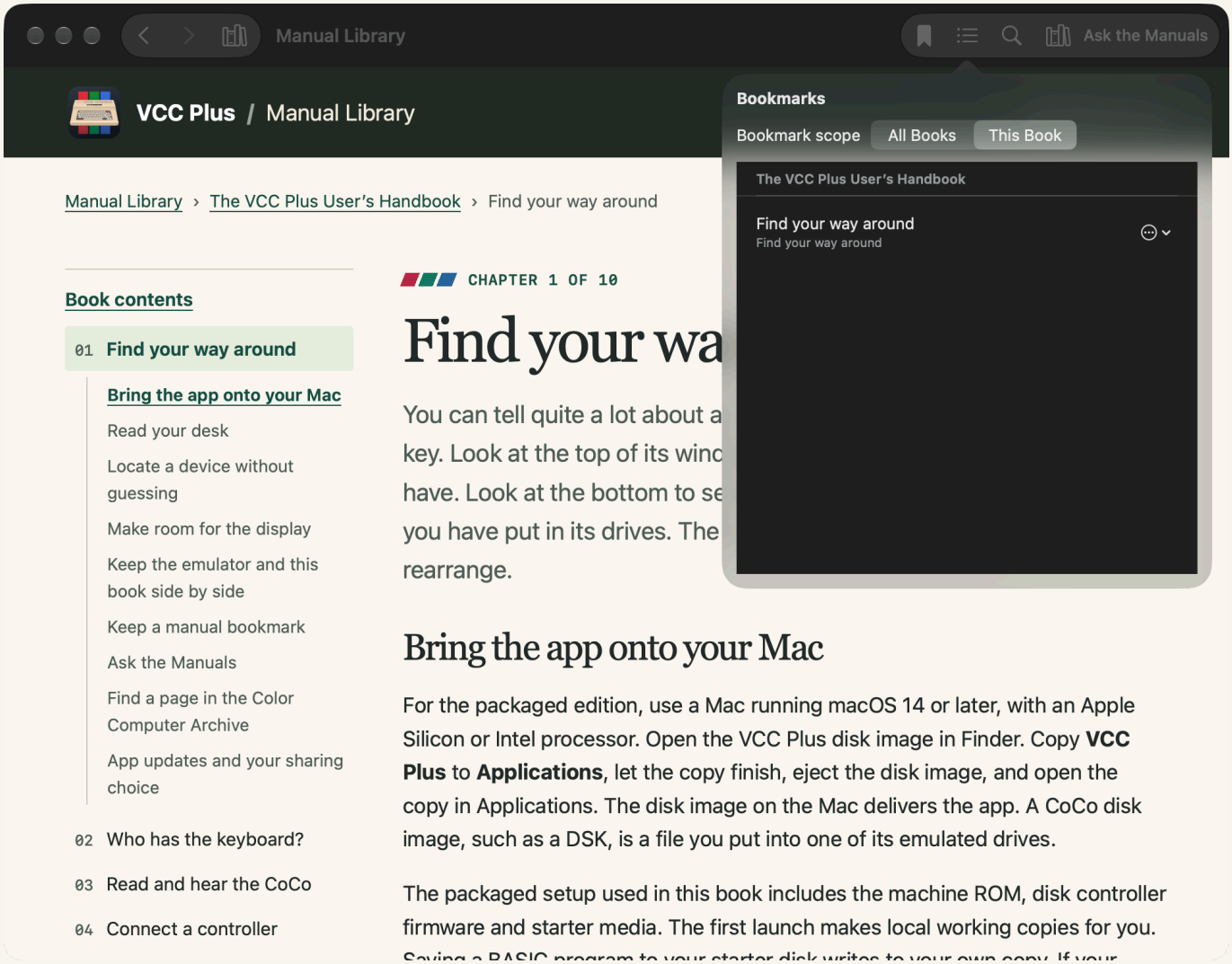


Figure 2: The Bookmarks popover offers All Books and This Book. Here This Book lists a saved location in the User's Handbook, with an actions menu beside it. The filled bookmark symbol identifies the saved reading location.

To find a word on the page you are reading, choose **Find on This Page** in the reader toolbar or press **⌘F**. The counter shows your place among the matches. Use **⌘G** for the next match and **⇧⌘G** for the previous one. Press **Escape** to return to the page.

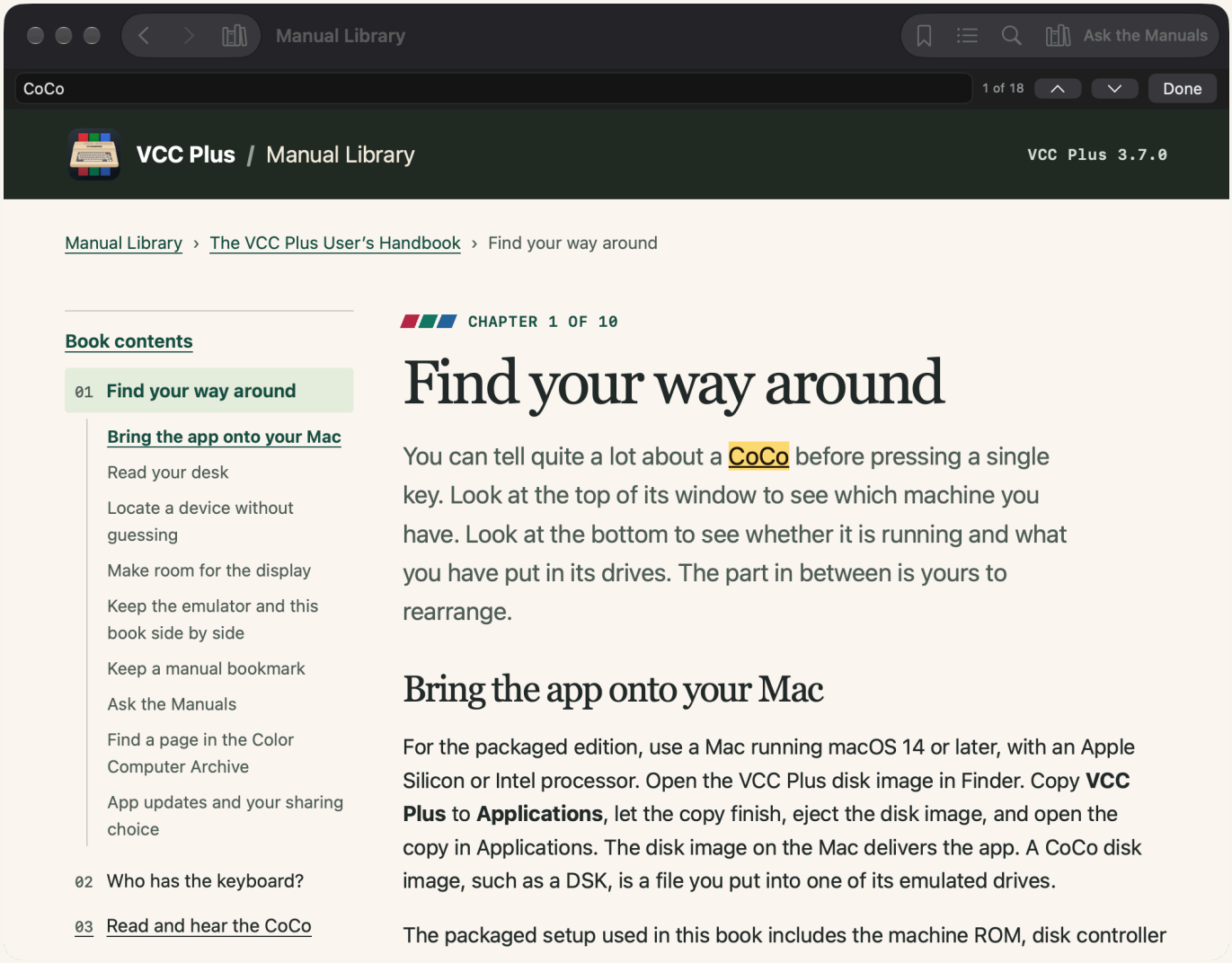


Figure 3: The magnifying glass beside Ask the Manuals opens page Find. Here the reader highlights the first of 18 matches for "CoCo."

The book window has its own keyboard focus. To send the next character to the CoCo choose **Input → Focus CoCo Display**. If you click the emulator's red close button while reading, choose **File → Show Emulator** (**Command-1**) to bring the same session back. A half-finished flight number makes a good way to prove this to yourself.

New to the green **OK** prompt? Get your first reply. Returning from a physical CoCo? Match the hardware to the Mac controls.

Ask the Manuals

Choose **Ask the Manuals** in the reader toolbar to search the twelve books written for VCC Plus and installed with the app. Type one or more keywords. Try `joystick keyboard` to find sections that mention both, or `PRINT@` to look up that exact command. Matching is case-insensitive and keeps punctuation in command names.

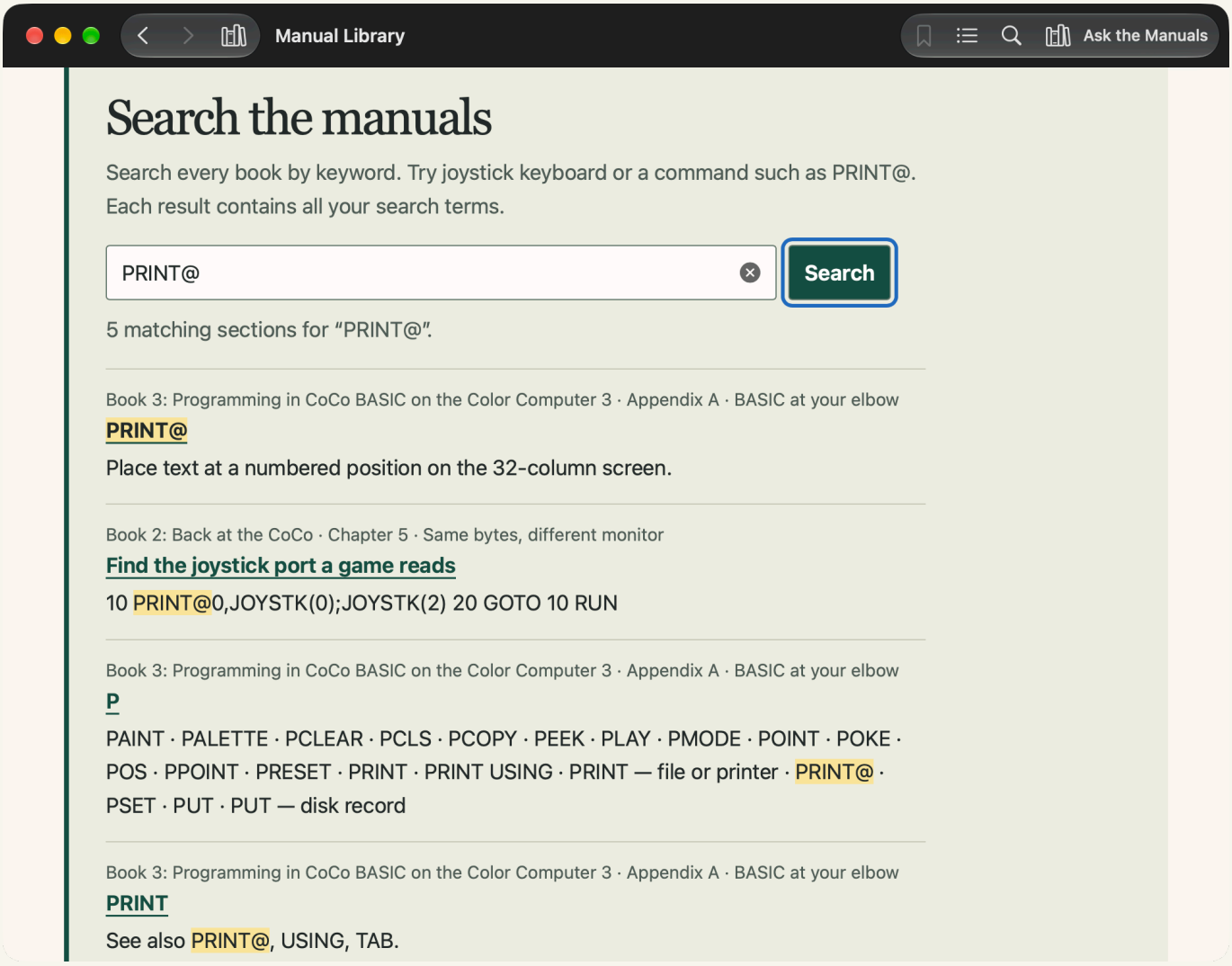


Figure 4: A PRINT@ search lists the command reference first, followed by examples that use it.

Select a section title to open it. Each result shows the book, chapter, and an excerpt with matching words highlighted. Choose **Back** to return to your results. The shelf in a browser uses the same offline search.

Find a page in the Color Computer Archive

Help → Archive Library... is the place to browse period manuals, magazines, diagrams and supporting documents. **Manual Library** contains the books written for VCC Plus; **Archive Library** searches the historical collection preserved by the Color Computer Archive. Reading either leaves the running CoCo and your development documents in place.

Browse a publication or search its pages

Leave **Search Archive** empty to browse. Choose **All Documents**, or a collection in the sidebar such as **Magazines**. Expand publication folders to see issues grouped by year where the filenames supply dates. At the bottom of the list, the previous/next result buttons move through further batches. In a narrow window, **Collections** opens the sidebar.

For a lookup across publications, choose **Entire Archive** in the scope picker and type a few words, such as `JOYSTK` or `MOTD strip buffer`. Results are matching pages or sections, grouped under their publication titles. To limit a search, choose the intended collection and select that collection in the scope picker. Typing a query does

not by itself open the first result. Click the page or section you want, or move the selection with the arrow keys. Clearing the query returns to the browse list.

A title opens with one click; no separate Read or Download button is required. If the file is not cached, the reader shows **Downloading...**, the amount received and verification progress. **Cancel** stops that transfer; **Try Again** retries a canceled or failed load. Verified cached originals and the installed index can be used offline. A first download still needs a connection and enough local storage. In compact view, **Browse** or **Results** returns from the reader to the list; Return can reopen the selected row.

Archive Library remembers your collection, search/scope, selection and useful list position after closing the window or relaunching. Each document also keeps its own reading position, including supported PDF zoom and text/image scroll. A search hit still opens the exact cited location; it takes precedence over an older remembered page. Clearing the query returns to your browse context. Reading history stays local and does not grant an assistant new access.

From Software Library, a title's Guides & References can bring you directly here. A Raaka-Tu walkthrough, for example, opens as text separate from the game's main manual. The source software selection and running CoCo remain in place.

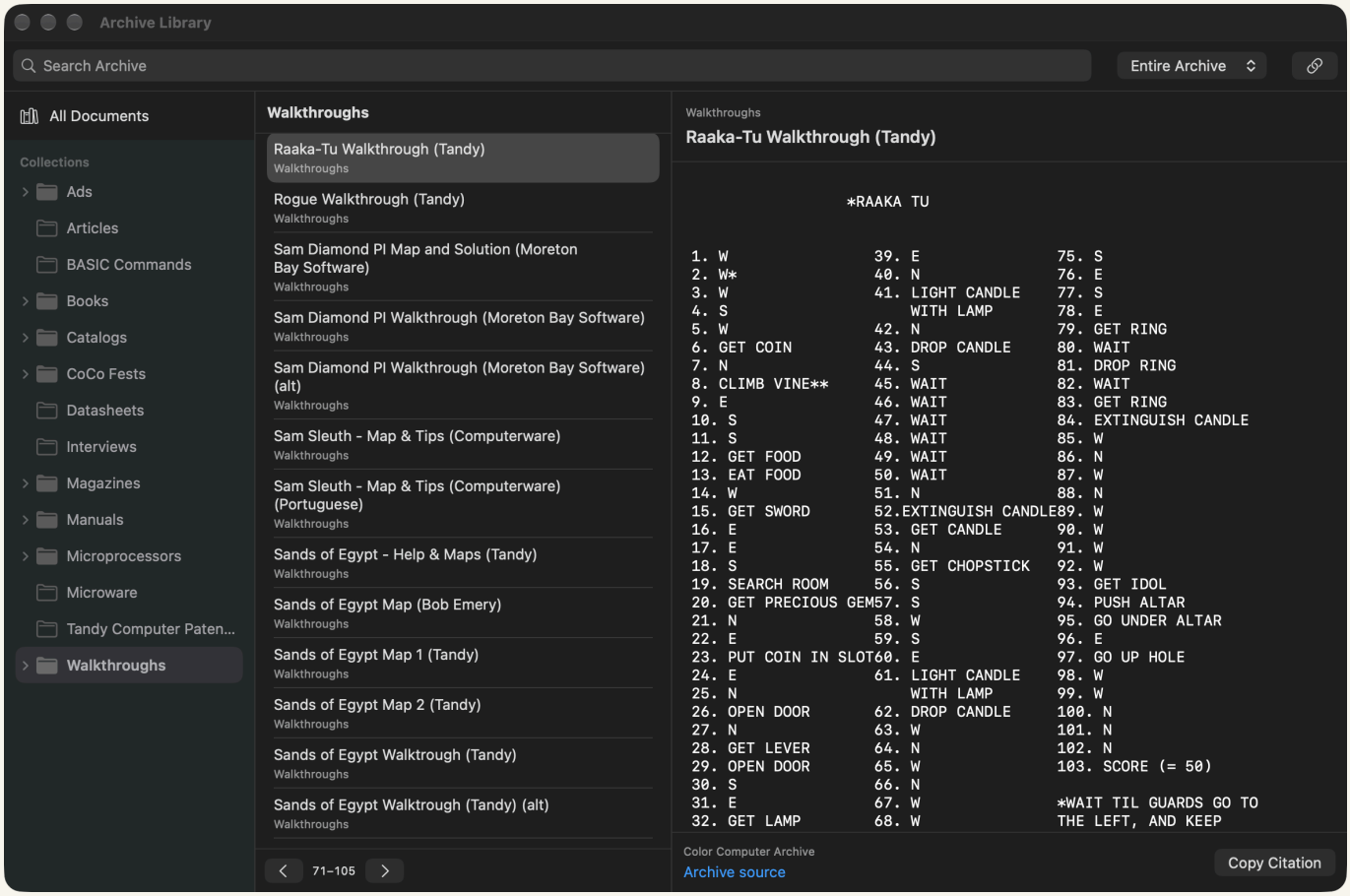


Figure 5: Raaka-Tu Walkthrough opens directly as text in Archive Library. Its Walkthroughs collection stays visible, and Archive source and Copy Citation retain the Color Computer Archive reference.

Read the kind of document you found

What opened	Controls to use
A PDF scan	Previous/next page, the page-number field, zoom and **Fit**. A search hit opens its cited physical PDF page. **Find in PDF** and **Find Next** search the PDF's own text layer; Command-F focuses the field.

What opened	Controls to use
A photograph, map or diagram	**Zoom in**, **Zoom out**, scrolling and **Fit Image**. Fitting changes the preview, not the original file.
Plain text, including a text listing	Select text and use the Mac's **Copy** command or Command-C. This is a read-only viewer; paste into an appropriate working document to edit it.
Indexed web, mail, word-processing or worksheet text	Read the extracted text in the app. Longer items have previous/next section controls and a section-number field. Worksheets retain row references and tabs when copied; this is not a live spreadsheet application.

Archive source opens the upstream document or parent archive. Use it when an edition cannot be displayed in the app, a cited PDF page cannot be opened, or you need the original layout rather than indexed text. A filename and a search excerpt alone do not prove a document is the edition you expected; check its title, collection, page and any ZIP member named above the reader.

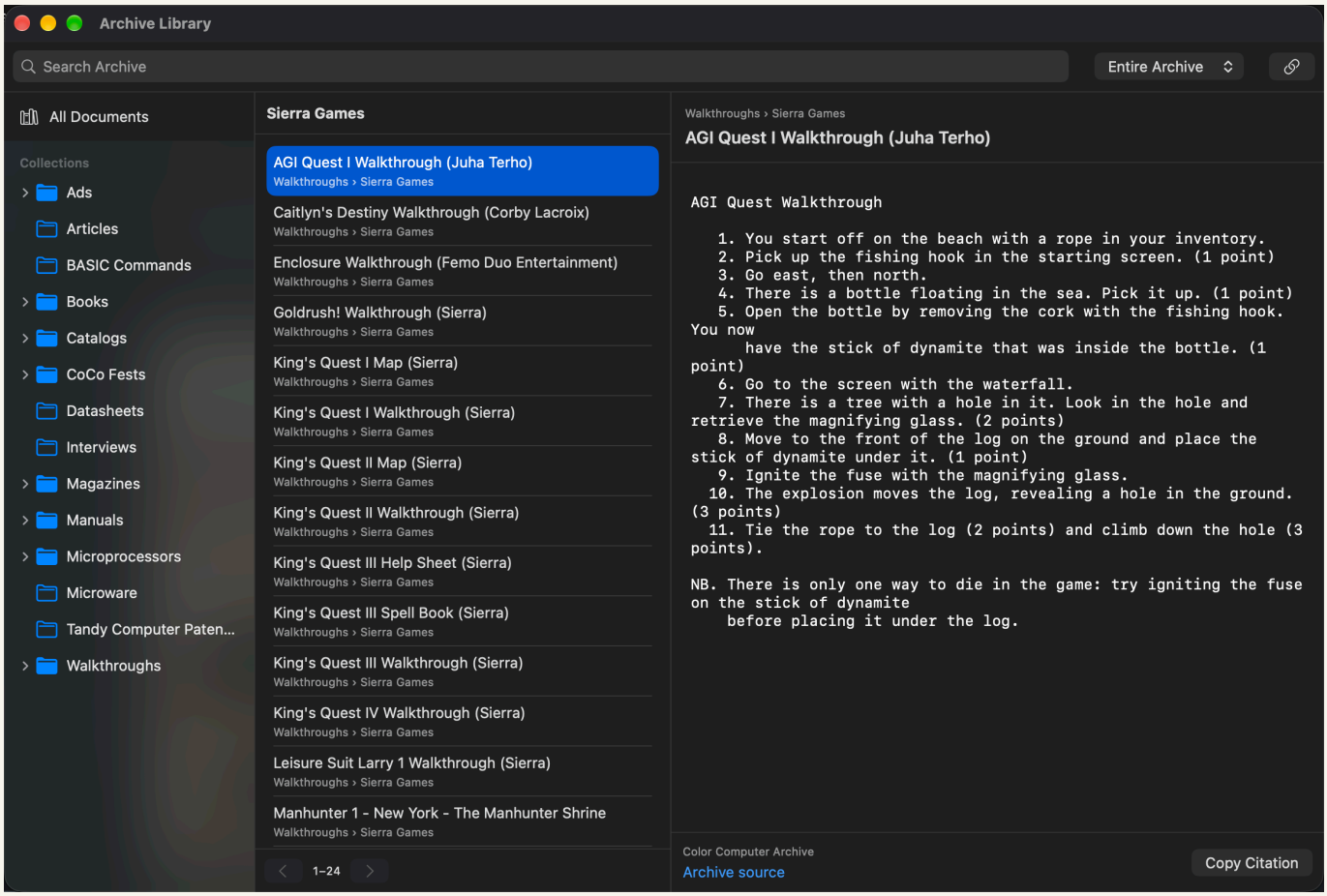


Figure 6: A walkthrough opens as selectable text inside Archive Library. It remains a historical reference rather than an editable guest program.

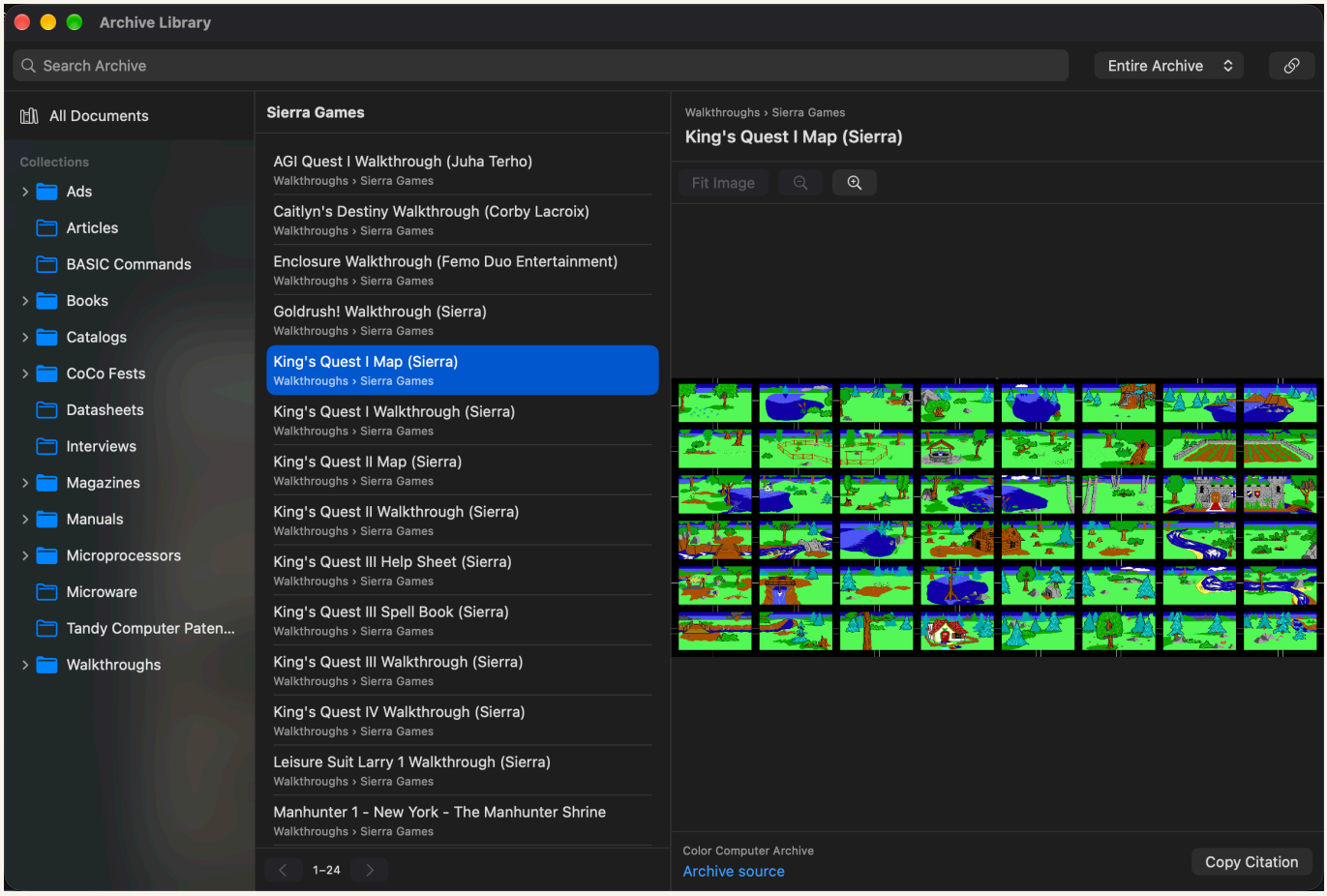


Figure 7: An archived Sierra map opens directly in the image reader with Fit Image and zoom controls.

Copy text without mistaking OCR for source code

An **OCR** match means the search index recognized words from a page image. It does not mean the displayed PDF has a selectable text layer, and it does not mean its program listing has been corrected. A search can find the right page even when **Find in PDF** reports no matching text. The image viewer itself has no OCR-to-code button.

If the PDF has selectable text, select the passage and press Command-C. Plain-text and indexed-text views also support selection and copying. Check copied program text against the scan: zeros and letter O, ones and letter I, quotes, punctuation, line wrapping and BASIC line numbers can be misread or rearranged. A selected paragraph, a search excerpt and a complete runnable program are different things. Preserve the original reference while you repair a working copy.

Use a reviewed example or snippet

On selected PDF pages, actions above the scan provide a reviewed source rendition associated with that exact publication edition and page range. These actions are available only where the index contains such a rendition. Their absence on another page is normal.

Action	What goes to your clipboard or workspace
Copy Snippet	The selected reviewed fragment as plain source text, including its stored line breaks. It can depend on definitions or surrounding code elsewhere in the article.

Action	What goes to your clipboard or workspace
Copy Example	The selected reviewed example's source text. Read its language and provenance before deciding where to paste it.
Open in BASIC Editor	A complete reviewed BASIC program in its own **BASIC Example** workspace. This action is offered for BASIC programs, not every listing.
Example source information or **Examples on this page**	The example title, language, author and provenance notes. When a page has several examples, choose the intended title before copying or opening it.

For a concrete fragment, search `MOTD strip buffer` and open **MOTD 1986 04**, **PDF page 4**. **Copy Snippet** supplies Strip's character constants. On **PDF page 12**, the buffer-setup snippet has different instructions and depends on definitions elsewhere in the listing. The provenance notes identify what was transcribed or checked; neither fragment is the whole Strip utility.

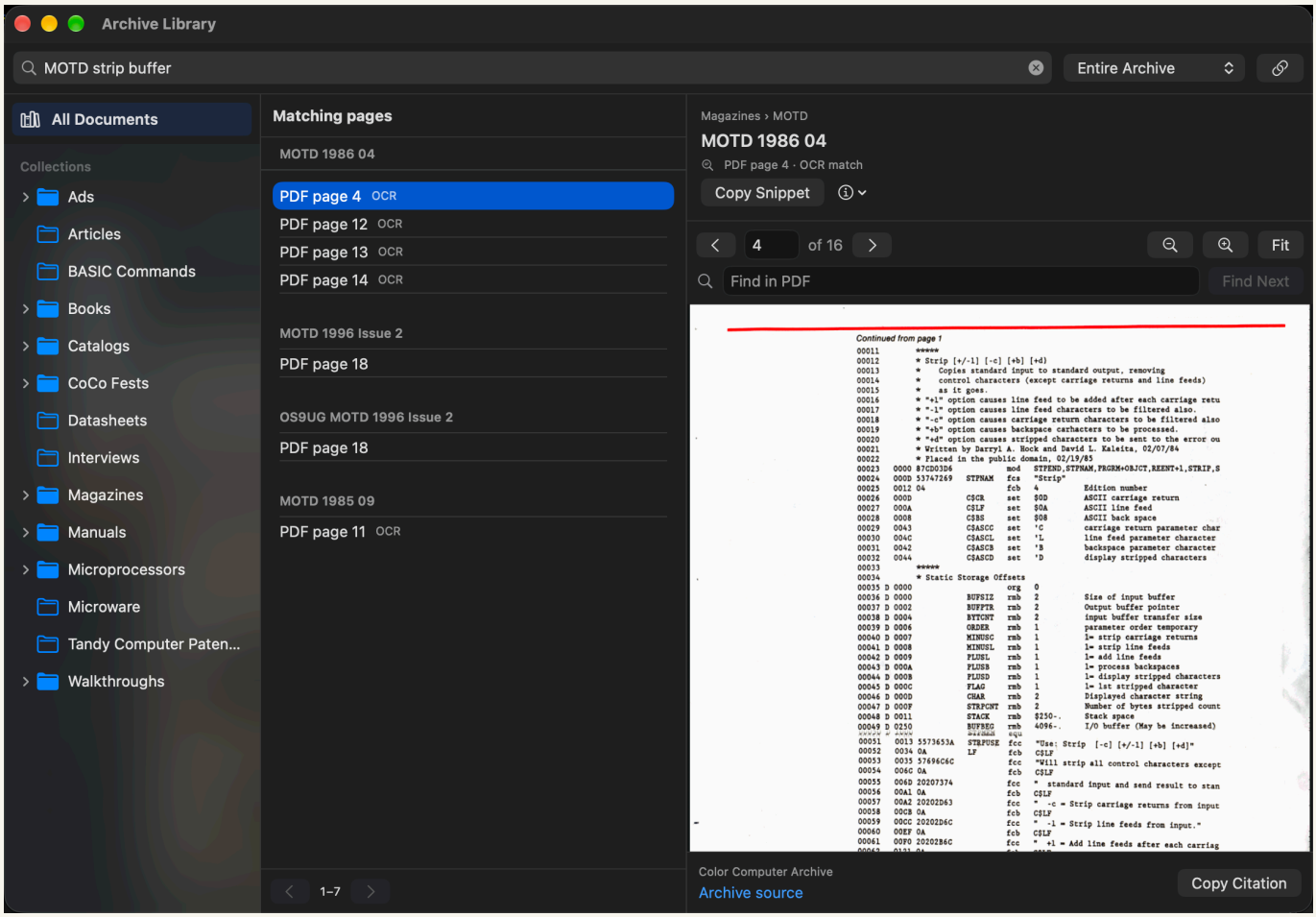


Figure 8: The Strip article opens at the cited physical PDF page. Copy Snippet and Example source information identify the reviewed source associated with that page; Copy Citation keeps the Archive reference.

For complete BASIC programs, the January 2005 *CoCo Nutz!* issue offers **Mini Data Base** and **Pill** on physical PDF page 13. Choose the title under **Examples on this page**, then **Open in BASIC Editor**. Each reviewed source has a separate, recoverable workspace. Reopening the same example can return to your existing edits or recovered draft. Use the editor's **Save** or **Save As...** to keep a named Mac source document.

Opening an example does not overwrite your main BASIC document, load the CoCo or execute the program. Inspect the source and the required environment first. **Load into CoCo** and **Run** are explicit editor actions. They validate the numbered source and retain the usual confirmation before replacing an unrelated guest BASIC program or taking over an assembly session. Save any program and variables you need before confirming replacement. The editor rechecks the source before transfer; if you changed it while deciding, review the current version again.

There are currently no direct **Open in C Editor** or **Open in Assembly Editor** buttons in Archive Library. For C or assembly material, copy the reviewed source or verified text, create or choose the intended document in the corresponding workspace, and paste there. Preserve an existing draft before replacing its contents. Add any required headers, symbols or surrounding project files deliberately; then compile or assemble and test through that workspace. Copying a listing does not configure its build or prove it runs on the selected processor.

A reviewed rendition records a specific review or correction, not a guarantee that every historical program works in every setup. If **Example information could not be read** or **This example is unavailable in this Archive index** appears, use the scan and its citation; do not substitute an unrelated listing just because its name is similar.

Keep the source of what you copied

Copy Citation is separate from copying code. It gives the publication, the current page or section, any ZIP member name and the original Color Computer Archive link. Keep the physical PDF page distinct from a printed page number. After pasting the source into your working document, copy the citation into your notes or an appropriate source comment, and keep the author/provenance from the example information as well. Copy Citation replaces the clipboard's previous contents.

The scan or asset may be delivered from VCC Plus hosting and cached locally. Its citation still identifies the Color Computer Archive as the source. Cached original bytes and a reviewed source rendition have separate identities: editing your example does not change the archived scan, and corrected code should not be described as an unchanged transcription unless its provenance says so.

Let an assistant search the Archive

The Archive window's connection control offers **Save Archive Search Connection…**. This separate, read-only connection permits Archive searches and citation lookup. It does not grant control of the CoCo or a development project, and finding a citation is not proof that the assistant read the scanned page or validated its code. Keep VCC Plus open, and use **Revoke Archive Access** when finished. For project-editing permissions, see Work with an assistant.

App updates and your sharing choice

VCC Plus → Check for Updates… asks the website whether a newer compatible download is available. **Updates & Privacy…** shows the current version and build, offers the same check, and lets you choose whether to check when the app opens. An available update offers a DMG download. Open that image and install the app yourself when your work is safely saved; checking does not replace the app or restart the CoCo. If the website cannot be reached, keep using the app and try again later.

The first check asks whether you will share basic platform information to help understand the types of Macs people use. **Share** includes macOS major/minor version, Apple Silicon or Intel processor family, primary language, configured region and a random installation ID. **Don't Share** leaves those optional details out. Either choice is remembered across app updates, and checks and downloads work either way. Your configured region is a preference, not a measurement of your physical location. The app does not send your documents, disks, assistant conversations, account name or hardware identifiers.

Change **Share basic platform information** in Updates & Privacy at any time. **Reset Installation ID** starts a new statistical identity; it does not erase earlier server history. The update service records checks and download-button selections in a private history. Website hosting and security logs are separate from optional app statistics.

Privacy Information opens the website's full description. DriveWire has equivalent commands and remembers its own sharing choice and installation ID independently.

Who has the keyboard?

Chapter 2 of 10

BASIC is asking for a flight number. You can see the question behind the source editor. If you press 4 now, will it answer the question or change the program you wrote? On a Mac, the place that receives your next keystroke has *keyboard focus*. You can move it deliberately.

Give the CoCo your next keystroke

Click inside the live CoCo picture, or choose **Input → Focus CoCo Display** (Option-Command-K). This brings the emulator forward and directs typing to the guest computer. On the toolbar, **Pause** means the CoCo is running. If the button reads **Run** and the status reads **Paused**, click Run before answering a guest prompt.

To operate the Mac controls, choose **Input → Focus Mac Controls** (Option-Command-U). The **Profiles** button receives focus. Press Space to open it, or use Tab and Shift-Tab to move between controls. If Tab skips buttons, check **System Settings → Keyboard → Keyboard navigation** on the Mac. Clicking inside a source document gives ordinary editing keys back to that editor.

The difference matters with Escape. In a Mac menu, Escape dismisses the menu. Inside the guest, it sends **Break**, which can interrupt a program. Command shortcuts belong to the Mac even while you're typing into the guest.

Finish an answer after reopening the window

Start in **Disk BASIC**. Save any BASIC program you want to keep. Follow KEYS.bas. In the app it opens in a separate **BASIC Example** window. In a browser, open the downloaded file in **Develop → BASIC Development…**. Choose **Run** in the window containing KEYS. Confirm replacement of the program in CoCo memory if prompted. Watch for the flight-number question.

1. Leave the question waiting and use the Mac's **Window** menu to return to the BASIC window you ran. In that editor you can see the numbered line that asks the question. Go to **Input → Focus Mac Controls**. Press Space to open Profiles, then close its window without loading a setup.
2. Choose **Input → Focus CoCo Display**. Type **4** and leave Return alone. Look for the digit on the question line.
3. Close the emulator window with its red close button. Keep VCC Plus open. Press **Command-1** (**File → Show Emulator**). You should see the digit you left behind.
4. Type **2** and press Return. The complete answer is 42. BASIC prints **FLIGHT 42** and returns to **OK**.



Figure 9: The emulator after reopening. The question contains 42, followed by FLIGHT 42 and OK.

Reopening the window brought back a running program halfway through an answer. A cold restart clears that memory, and quitting ends the session. If you've written something you want tomorrow, save the source or the guest program before quitting.

Match your keyboard to the program

Open **Input → Keyboard and Joysticks…** and find **Keyboard mode**. Changes take effect immediately and are remembered.

On a fresh installation the right joystick uses the sole connected controller, or Mouse if there is no unambiguous controller; the left joystick starts at None. This keeps ordinary typing available. A controller connected later can replace the fresh Mouse fallback until you make your own right-port choice. Existing saved input choices are preserved. See joystick sources to choose another source or port.

Mode	When to try it
Natural	Enter a BASIC listing or a letter. The printable keys follow the characters you type on your Mac layout, including punctuation.
CoCo	Follow a program's illustrated keyboard or a game that reads key positions. VCC Plus uses physical US CoCo positions, so punctuation can differ from the symbol printed on your Mac key.
Compact	Use character-based typing on a small keyboard. Tab sends **Clear** and the unshifted grave accent key sends **Break**.

Return is CoCo **Enter**, Home is **Clear**, and Escape is **Break**. F1 and F2 reach the emulated function keys. Hold Fn or Globe if macOS otherwise uses those keys for hardware controls. **Help → Keyboard & Shortcuts…** lists the modes, key equivalents and Mac shortcuts together. If letters such as Z or X disappear only inside the CoCo, check whether a keyboard joystick has claimed them.

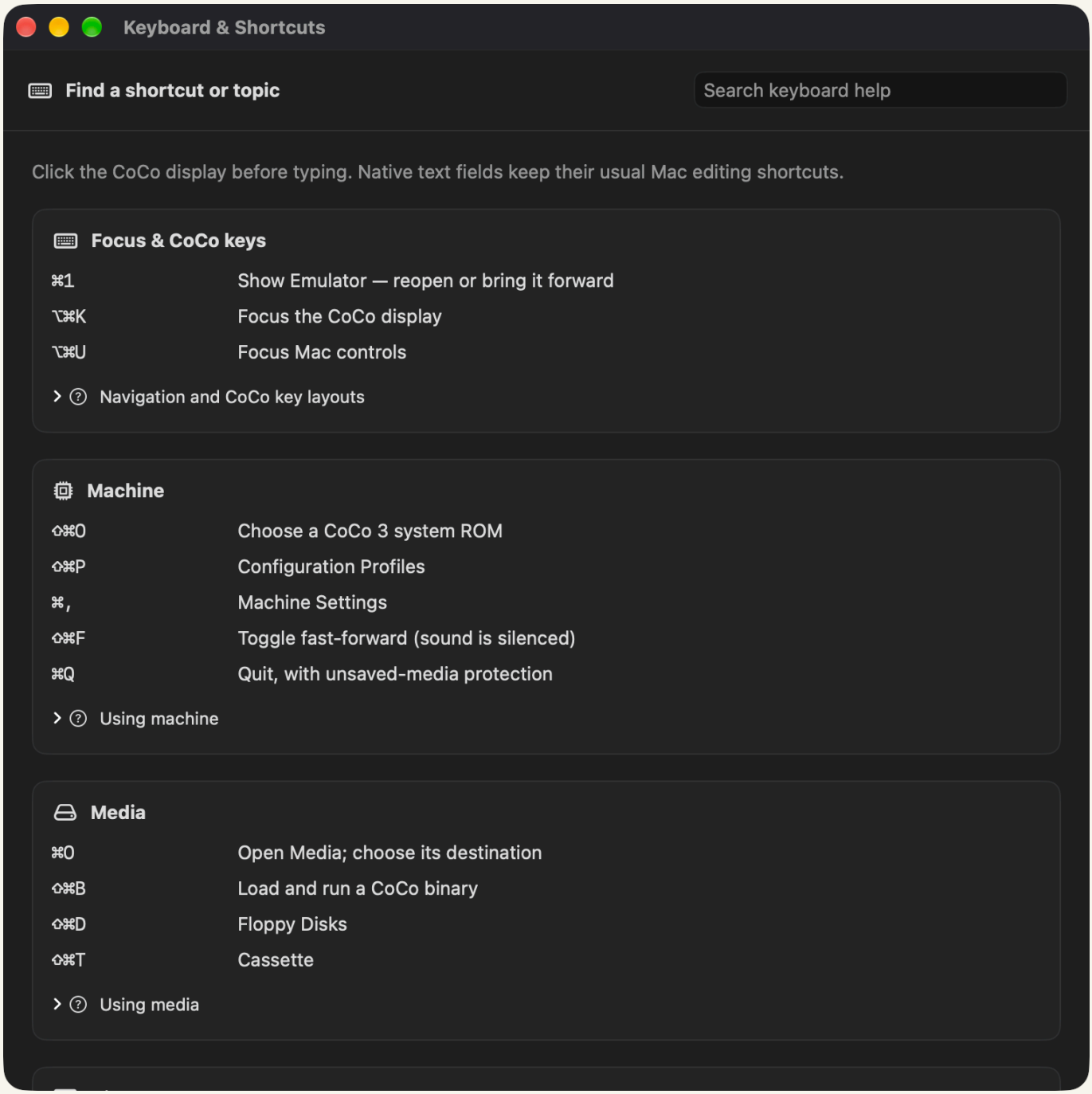


Figure 10: Keyboard and Shortcuts shows Mac focus commands beside the keys the guest receives.

Move text between the two computers

Choose the command for the place the text is going. Ordinary **Copy** and **Paste** in the source editor edit a Mac document. The separate commands under **Edit** communicate with the running CoCo.

What you want	What to choose
Take a guest message back to the Mac	**Copy CoCo Screen Text** (Shift-Command-C) copies the characters from a supported hardware text screen while running or paused. Paste into a Mac text field to read them. For a drawn game screen, save its picture.
Type clipboard text into a guest prompt	**Paste Text into CoCo** (Shift-Command-V) sends keystrokes. A newline in the clipboard sends Return, so check for extra lines before pasting at a command prompt.
Add numbered lines at BASIC's OK prompt	**Paste BASIC — Merge…** enters the numbered lines. Matching numbers replace existing lines. A number by itself deletes that line.
Start a fresh listing at BASIC's OK prompt	Save your old work, then choose **Paste BASIC — Replace…**. Confirmation sends NEW before the numbered lines. Copies previously saved to files survive.

The two BASIC paste commands leave you at the prompt so you can examine the listing before running it. **Edit → Cancel Paste**, pausing the emulator, or typing a CoCo key stops the remaining queued input. Already-entered lines have changed BASIC memory. For a program you want to edit and run repeatedly, keep a Mac source document in **BASIC Development…**.

VoiceOver can read the native menus, controls and media status. The **Devices** list in System view offers named controls alongside the drawing. For guest text, try **Copy CoCo Screen Text**; a game that draws its own lettering supplies pixels rather than hardware text characters.

Read and hear the CoCo

Chapter 3 of 10

Two pilots can run the same game, in the same kind of computer, and see different colors. The game draws a pattern in memory. The video signal decides how that pattern becomes a picture. Then the Mac decides how large to draw it. Here's how to tell which choice you're making.

See what the video signal changes

Choose a signal in **Display** on the Mac menu bar, in **Display → Display Settings…**, or under **Settings → Video Signal** in the main window. In System view the strip of bezel just below the monitor picture also opens the signal choices. Write down the current choice before trying another. A profile or a Software Library setup can select a signal when it starts a title.

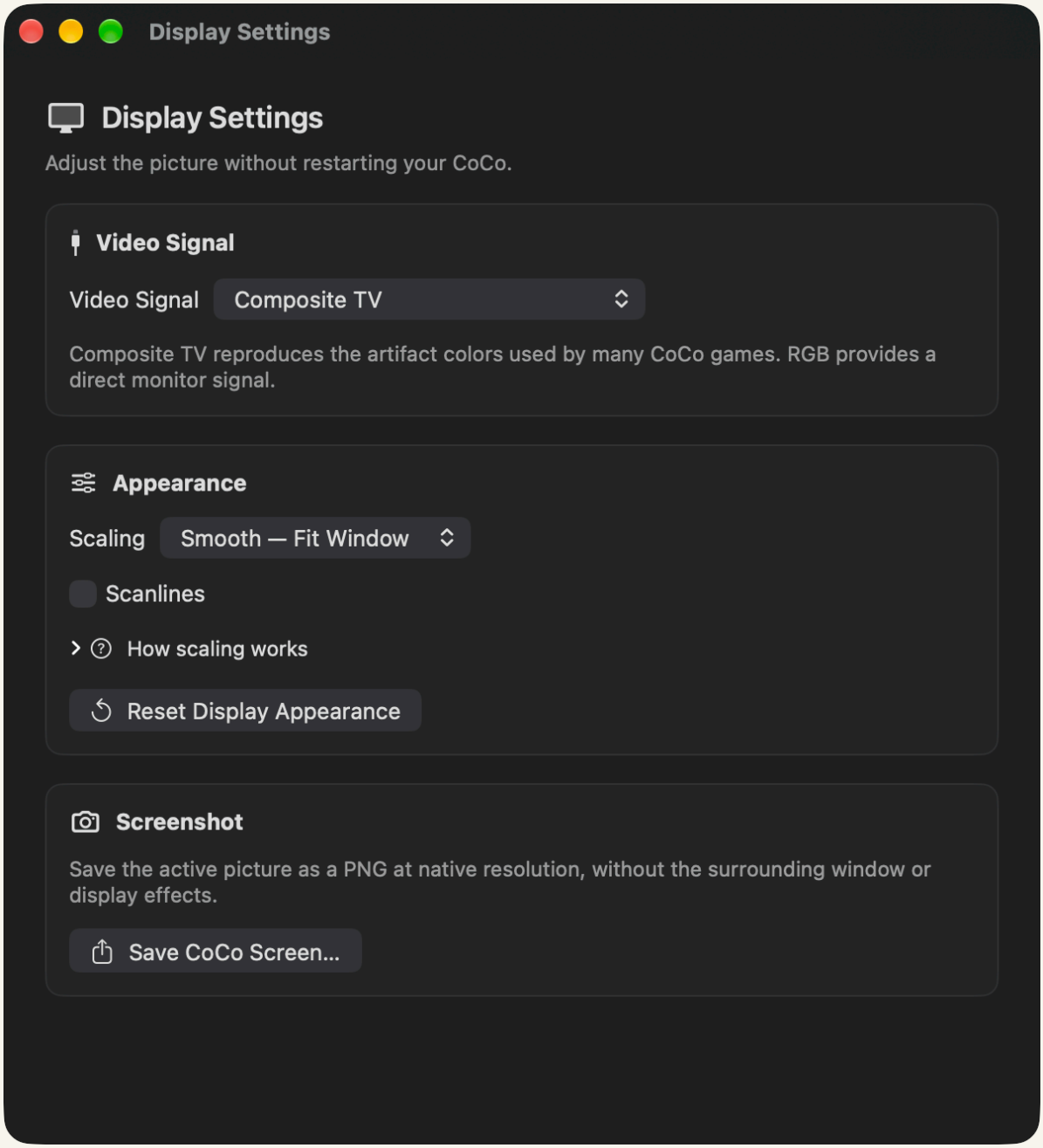


Figure 11: Display Settings brings the video signal and the Mac's display controls into one window.

RGB Monitor carries separate red, green and blue signals, as a CoCo 3 RGB monitor did. **Composite TV** recreates a TV picture in which very fine alternating bright and dark marks can appear colored. This is called *artifact color*. Some games drew their scenery with those patterns specifically to get colors from a TV. **Composite TV – Alternate Colors** reverses the artifact-color phase. Try it if you remember a familiar game's blue and orange the other way around.

Start in **Disk BASIC**, with your previous program saved. Follow LANDPAD.bas to open the game in a separate BASIC example window. If reading in a browser, open the downloaded listing in **Develop → BASIC Development...**. Choose the editor's **Run** and confirm replacement of the guest program if prompted. Your mission is to land under the beacon.

1. Choose **Display → Conventional Display**, then **Display → Composite TV**. Choose **Input → Focus CoCo Display** and press Space. Wait until two pads and a cross have finished drawing. Leave the landing keys, L and R, alone.

2. The left pad looks orange and the right one blue. Click **Pause** in the emulator. Choose **Display → Save CoCo Screen...** and name this picture **Composite.png**.
3. Choose **Display → RGB Monitor**. A paused CoCo still shows its last completed picture. Click the emulator's **Run**, give the CoCo a moment to draw, then **Pause**. The game waits for a landing key, so you can do this without losing your place. Save this picture as **RGB.png**.
4. Compare the pads. Both are made of bright vertical lines with one dark pixel between lines. The lines on the right begin on the opposite beat. On RGB you can see the stripes themselves. The beacon stays over the right pad in both pictures. Open the full-size images if the stripes blend at your reading size.

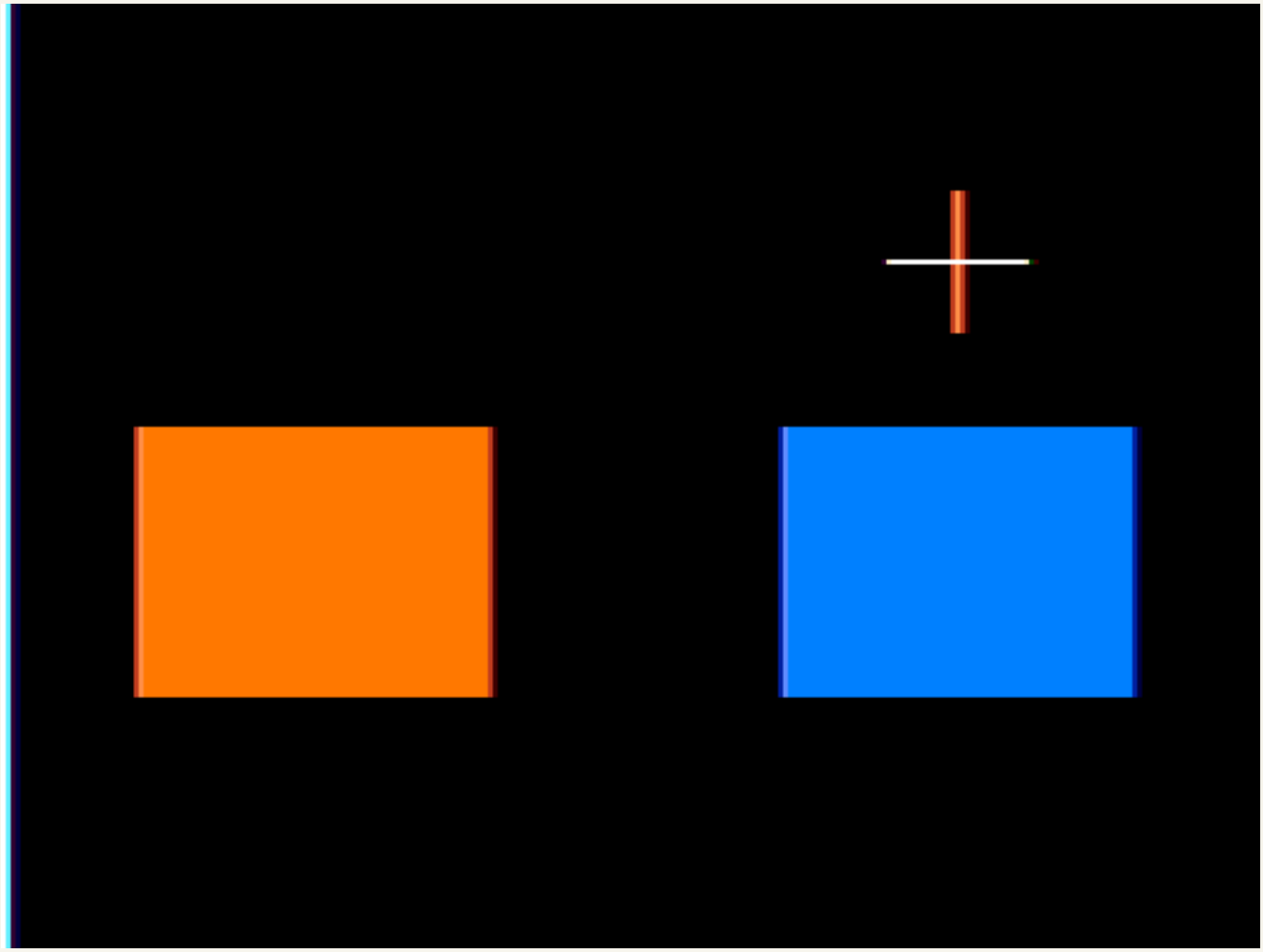


Figure 12: Composite TV. A cross above the blue right-hand pad marks the landing site. The left pad looks orange.

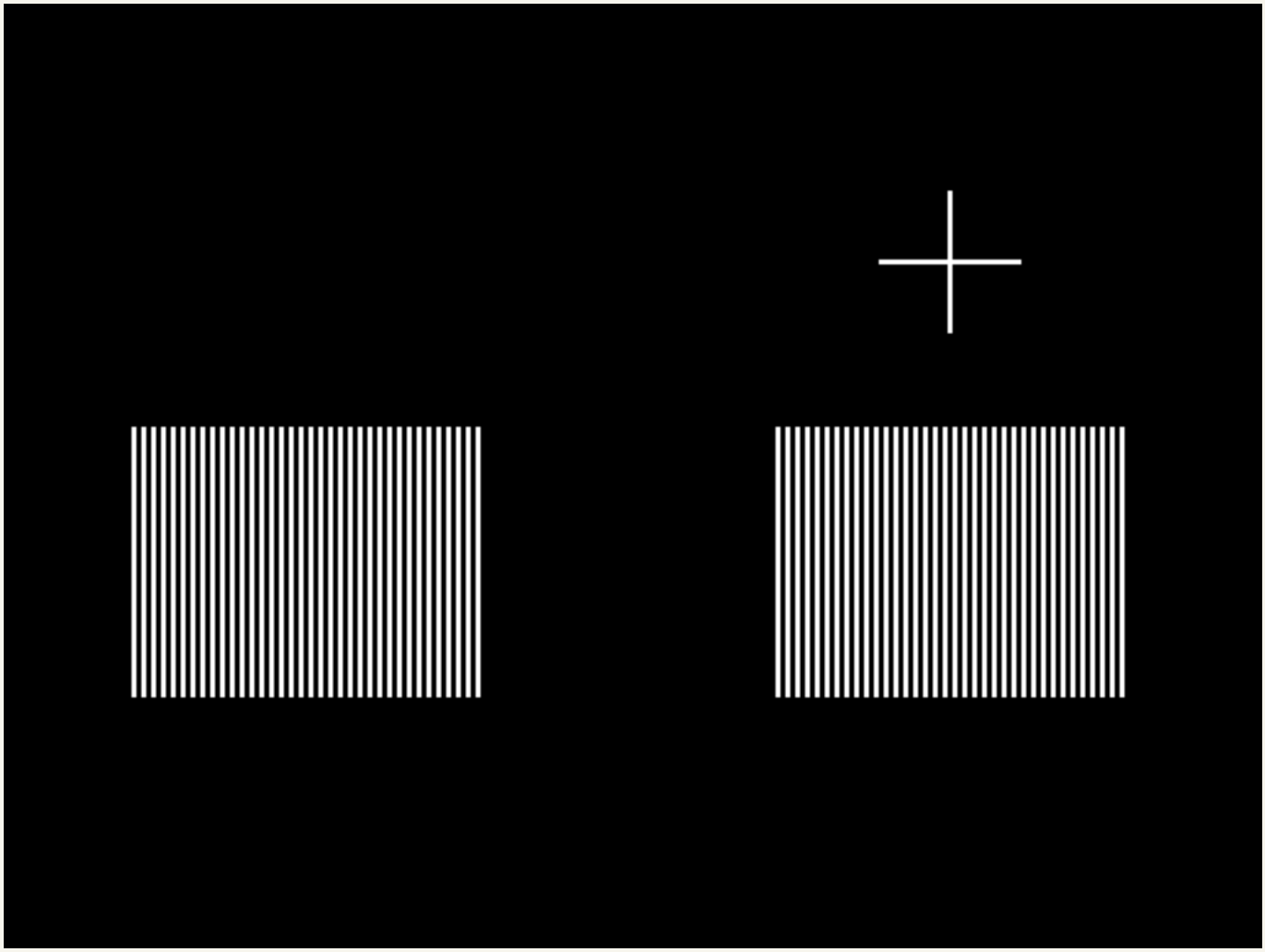


Figure 13: RGB Monitor. The same two pads show alternating bright and dark stripes, with the cross still on the right.

Give the monitor more room

Use **Display → Expand Monitor** in the illustrated System view to bring the live monitor forward in the same window. **Return to System** puts it back. **Conventional Display** gives the picture its own central area. All three presentations show the same machine. Drag a window corner to try a wider or taller shape. VCC Plus fits the picture while preserving its proportions; the extra space can land beside or above the picture.

Display → Display Settings... has three **Scaling** choices. Try them while the landing game is paused on RGB.

Choice	Look for
Smooth – Fit Window	The picture grows to fit. Softened pixel edges can make text and diagonal lines gentler at an in-between size.
Sharp – Fit Window	The same fit, with crisp pixel edges. Use this to inspect a game sprite or a tiny pattern.
Integer Pixels	Each source pixel gets a whole-number enlargement when there is room. A larger surround can appear because the next whole step would be too big. If even the original size cannot fit, the picture scales down proportionally.

Turn **Scanlines** on to add a horizontal line effect. It fades at small sizes to keep lettering readable. To compare settings precisely, hold the window size steady as you change one control at a time.

Save the CoCo picture or the Mac window

To take a picture, click the **camera icon** in the open header area between the machine nameplate and processor badge. Hover to see **Capture CoCo Screen**. The app saves a PNG directly to **Desktop** by default and shows a saved indicator with **Show in Finder**. You can keep playing rather than answering a Save dialog for every picture.

The shared disclosure beside the camera and video buttons has a **Screenshot Timer**: **None**, **5 Seconds** or **10 Seconds**. Choosing a timer changes the setting; it does not take a picture. Click the camera to use that timer. None captures immediately. A delayed capture shows its remaining seconds and **Cancel**. Keep using the CoCo until the timer finishes; the picture comes from that later moment, accompanied by a brief shutter sound. The timer is remembered for subsequent captures and launches. A guest reset or machine replacement cancels a pending timer, while switching Display/System presentation does not.

Save Screenshots and Videos To offers **Desktop**, **Documents** and **Other Location…**. Both kinds of capture use this one remembered folder. You can also change it in **Capture Settings…**. Filename collisions get a suffix rather than replacing an earlier capture. If saving fails, the captured picture remains available in Capture Settings for **Save Captured Image As…** or an explicit discard.

Display → Save CoCo Screen… remains the deliberate command with a native Save dialog. It freezes the picture at the moment you choose the command; the CoCo continues running while you choose its destination.

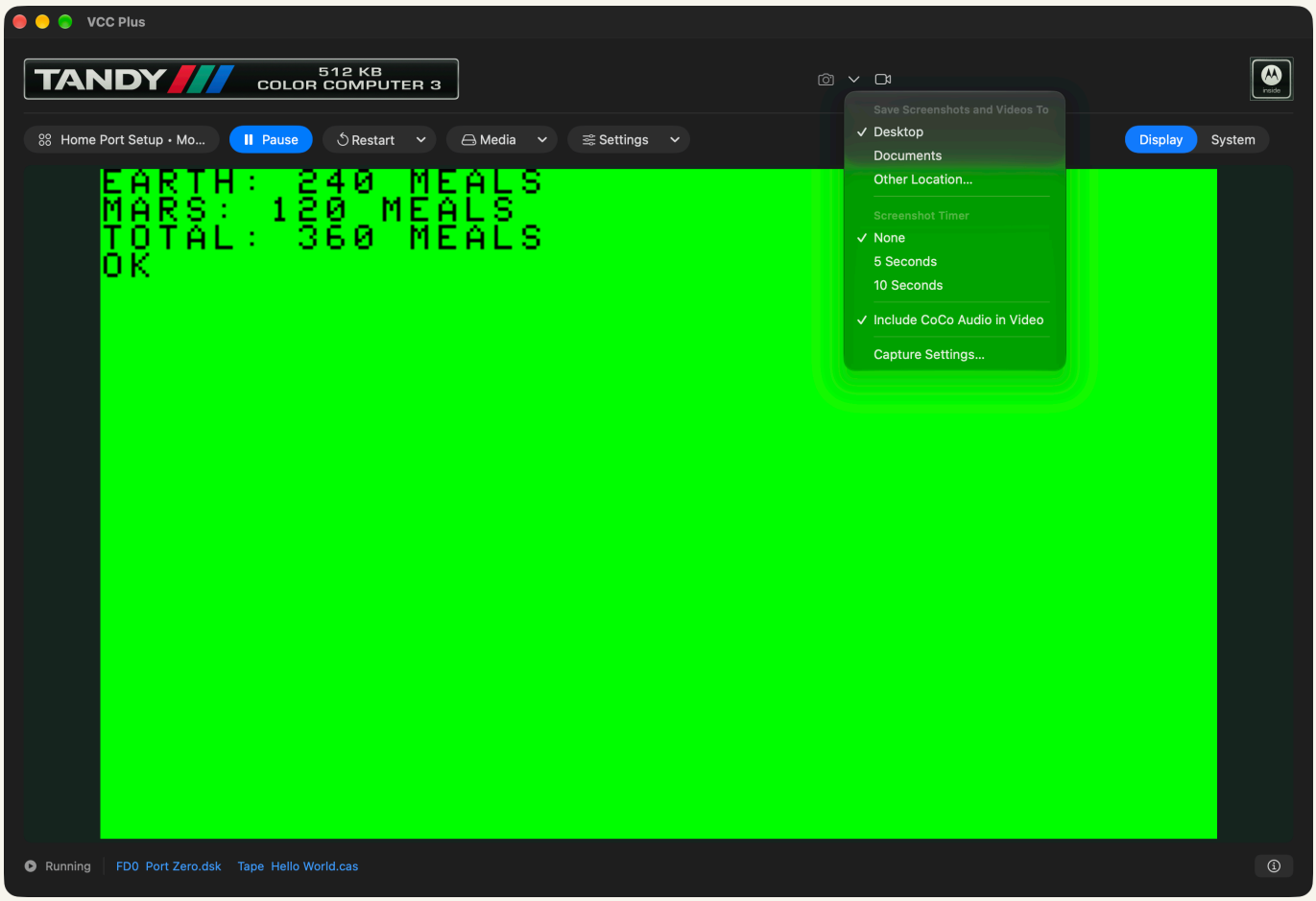


Figure 14: The shared capture options menu selects Desktop, Documents or Other Location for both screenshots and movies. Screenshot Timer offers None, 5 Seconds and 10 Seconds; Include CoCo Audio in Video controls the next movie. The camera and video buttons remain separate actions.

While still paused, change to **Integer Pixels** and choose **Display → Save CoCo Screen…** (Shift-Command-S). Save **RGB-Integer.png**. Open this and RGB.png in Preview at the same zoom. Both files in this exercise are 512

× 384 and their pixels match. Each is the active picture that came from the CoCo. Changing Mac scaling, the size of the window or the scanline effect changes how you see that picture on the Mac.

Both routes save **only the actual CoCo picture**. System artwork, the monitor case, toolbar, pointer and Mac desktop are excluded, even while System view is selected. A PNG retains the active picture's native raster dimensions; it does not bake in Mac window scaling or scanline effects.

To show someone a setting, a drive light, or the border around an integer-scaled picture, take a **Mac window screenshot** instead. Press Shift-Command-4, tap Space, then click the window you want. To share a text message for someone to read or search, try Copy CoCo Screen Text.

Finish the landing. Restore your scaling, choose **Composite TV**, click **Run**, and focus the CoCo display. Press **R**. The game prints **BEACON FOUND!** followed by **OK**. Pressing L instead gives **DARK PAD. TRY THE OTHER ONE!** and returns to BASIC. Restore your previous signal and presentation when you're done. For a closer look at the drawing trick, see Same bytes, different monitor.

Record the CoCo picture and sound

The separate **video icon** beside the camera starts **Record CoCo Video**. It records only the CoCo picture, with optional generated CoCo sound. System view, its hardware artwork, Mac windows, desktop and microphone are never part of the movie. The same controls are available through the Display menu.

A brief tape-start cue confirms that recording has begun. During recording, the icon becomes a red stop indicator and the elapsed time stays visible. Its hover explanation changes to **Stop Recording**. Reduced Motion uses a steady indication rather than the repeating pulse. Click Stop when your demonstration is complete. **Finishing…** means the app is completing the file; wait for the saved result before opening it.

The result is an **MP4** in the same remembered **Save To** folder as screenshots (Desktop by default). It uses H.264 video and stereo AAC sound. The fixed video canvas fits the current CoCo picture without cropping it; changing a guest video mode or resizing the Mac window does not record the surrounding UI. You can take a PNG while a movie is recording.

Include CoCo Audio in Video is on by default. Turn it off in the shared options or Capture Settings for a silent movie; the choice applies when the next recording starts. When it is on, the movie includes CoCo sound **even if local playback is muted or turned down**. The shutter and recording-start cues are Mac feedback only and never appear in saved movies or MCP audio/video captures. Your Mac speaker volume does not set the recorded audio level. A paused guest leaves a held picture and silence in the movie; choose Run to continue the guest. Ordinary soft/cold resets remain in the recording. Use normal emulation speed: fast-forward and recording are separate choices.

Recording keeps working files while the movie is being made and needs temporary space to complete the MP4. If a recording is interrupted or cannot be completed, open **Capture Settings → Retained Recordings**. **Recover MP4** tries to complete the retained movie, and **Show Files** locates its working data. Recovery may contain only completed fragments; do not delete those files while investigating a failure. A successful MP4 completion permits the app to remove its corresponding working file. Closing the app or replacing its machine while recording requires an explicit stop/save decision. Cancelling that decision keeps the current session.

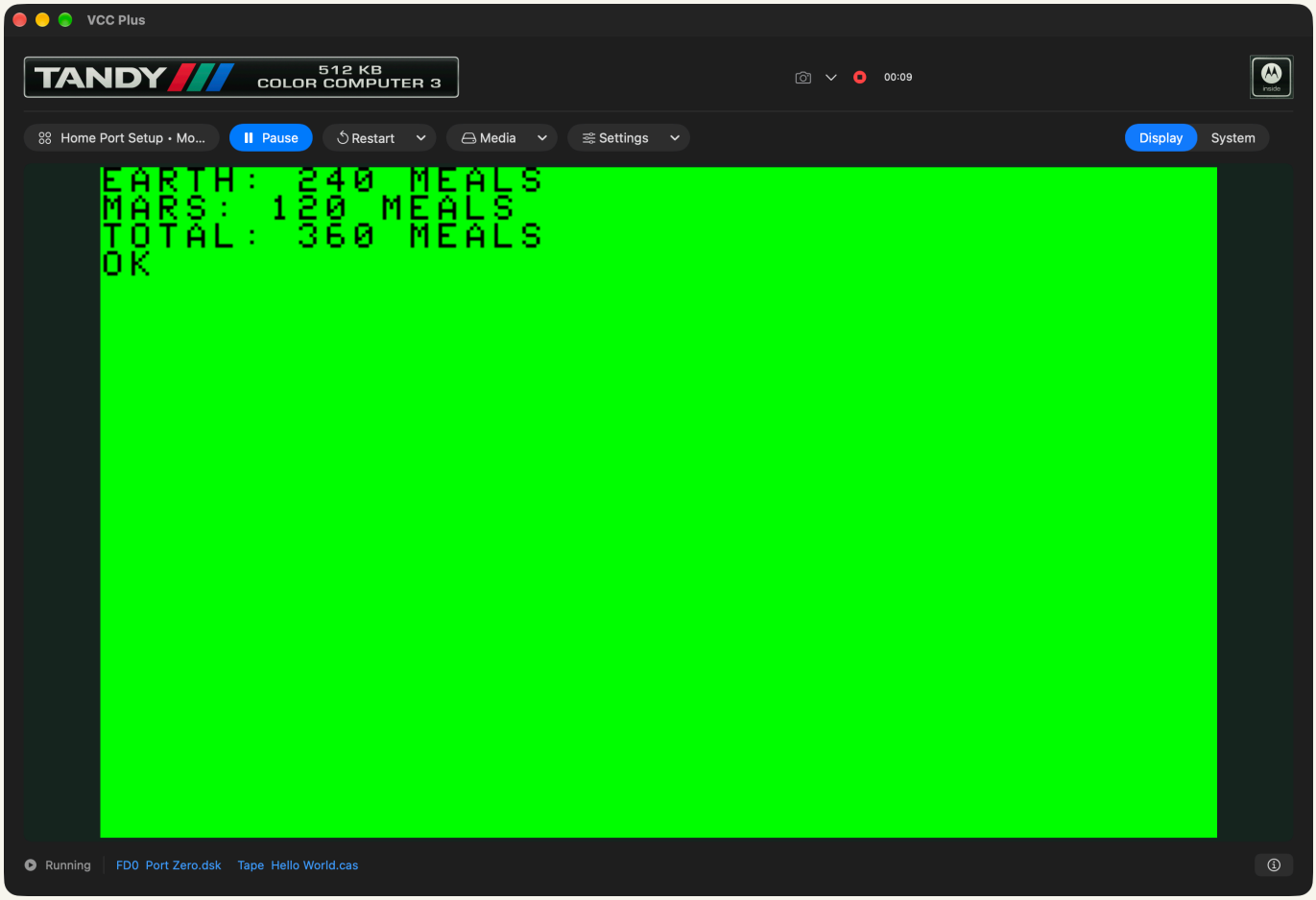


Figure 15: The video control becomes a red Stop button while a movie is recording, with the elapsed time beside it. The camera remains available for a still picture. The saved movie contains the CoCo picture rather than this Mac window.

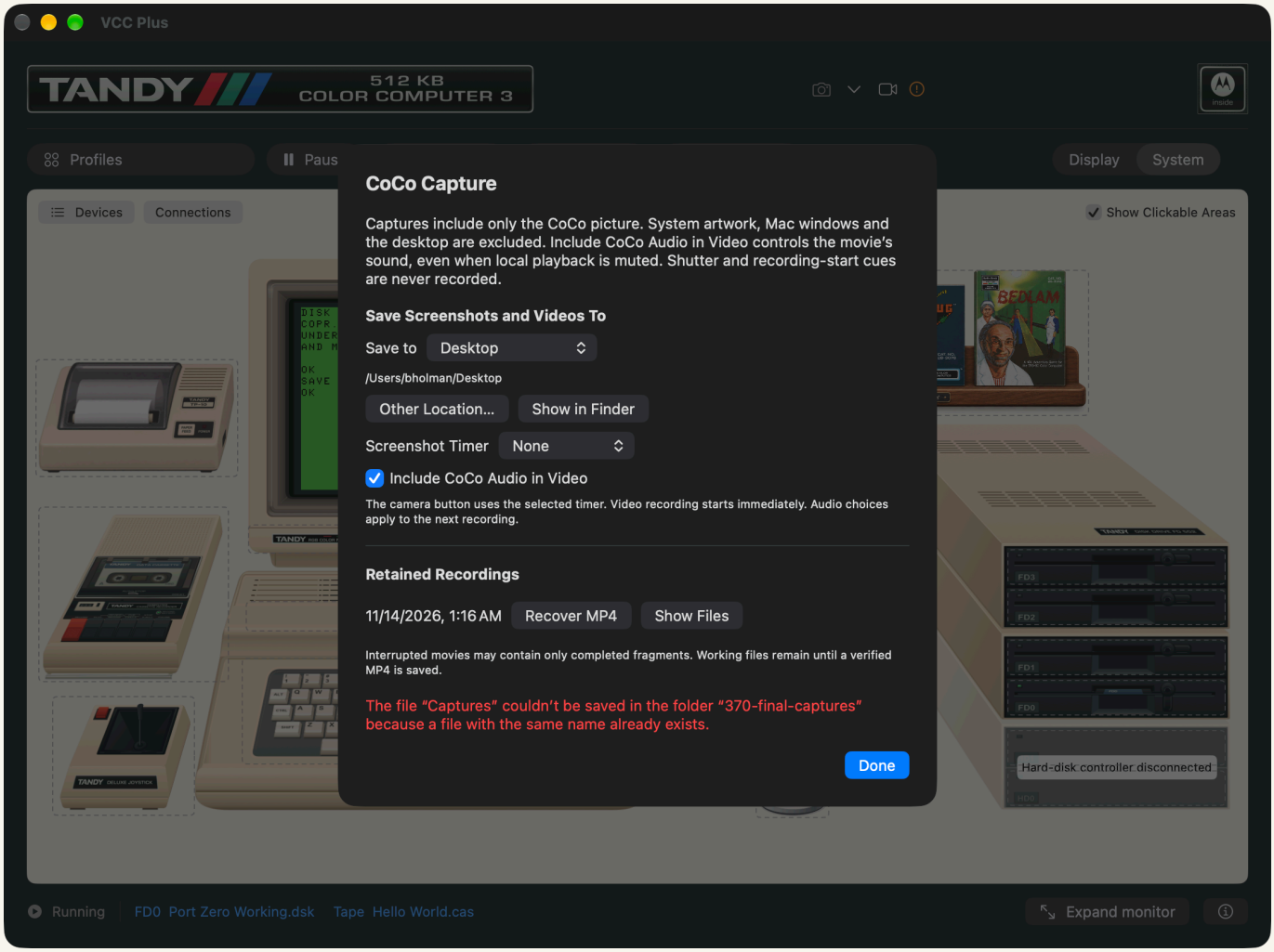


Figure 16: Capture Settings shares the save folder, screenshot timer and video-audio choice. This failed-export example retains the movie and offers Recover MP4 and Show Files; restore storage access before retrying.

Follow a sound from CoCo to Mac

Start with the state shown below the emulator. **Paused** means the machine has stopped executing. **Fast-forward • Sound silenced** means the CoCo is running ahead of real time with playback suppressed; tape and disk operations can still progress. Use **Machine → Fast-forward** to return to normal speed.

Open **Media → Cassette...** and look for **App sound**. Despite its home in the cassette window, this controls all CoCo playback, including a game or sound cartridge. Turn **Sound** on and raise **Volume**. At zero the label reads **Muted**. A moving game can therefore be silent while the emulator keeps running. Closing the cassette window preserves the tape transport and any recording. Use the tape's **Stop** if you mean to stop the recorder.

For a known example with both visible and audible results, see Give the pilot an arrival sound. The landing score in that exercise plays three notes and the program advances to its log. If you prefer a visual check, follow the landing text and log, and read the Sound switch, volume percentage and run state. Those checks tell you what the program and app are doing. To check the actual speaker path, listen to a known sound and check the Mac's selected sound output and system volume.

Connect a controller

Chapter 4 of 10

A game can wait at its opening screen even while you are moving a perfectly good joystick. There are two places to look. First, does VCC Plus see movement from your Mac device? Second, is that movement reaching the CoCo port the game reads? You can check both without guessing or restarting the game.

Choose the port the game reads

Open **Input → Keyboard and Joysticks…**. In System view you can also click the illustrated joystick, or choose it from **Devices**. The input window shows **Left CoCo joystick** above **Right CoCo joystick**. These are the two sockets on the emulated computer. A controller's own left and right thumbsticks are parts of a single device. Look for a port in the game's instructions, readme or manual. Choose the input source beside *that port*. **None** leaves it centered with buttons released. **Keyboard**, **Mouse** and **Controller** give you three ways to send movement into the socket. Changes apply immediately and are remembered. Note any previous assignment so you can put it back.

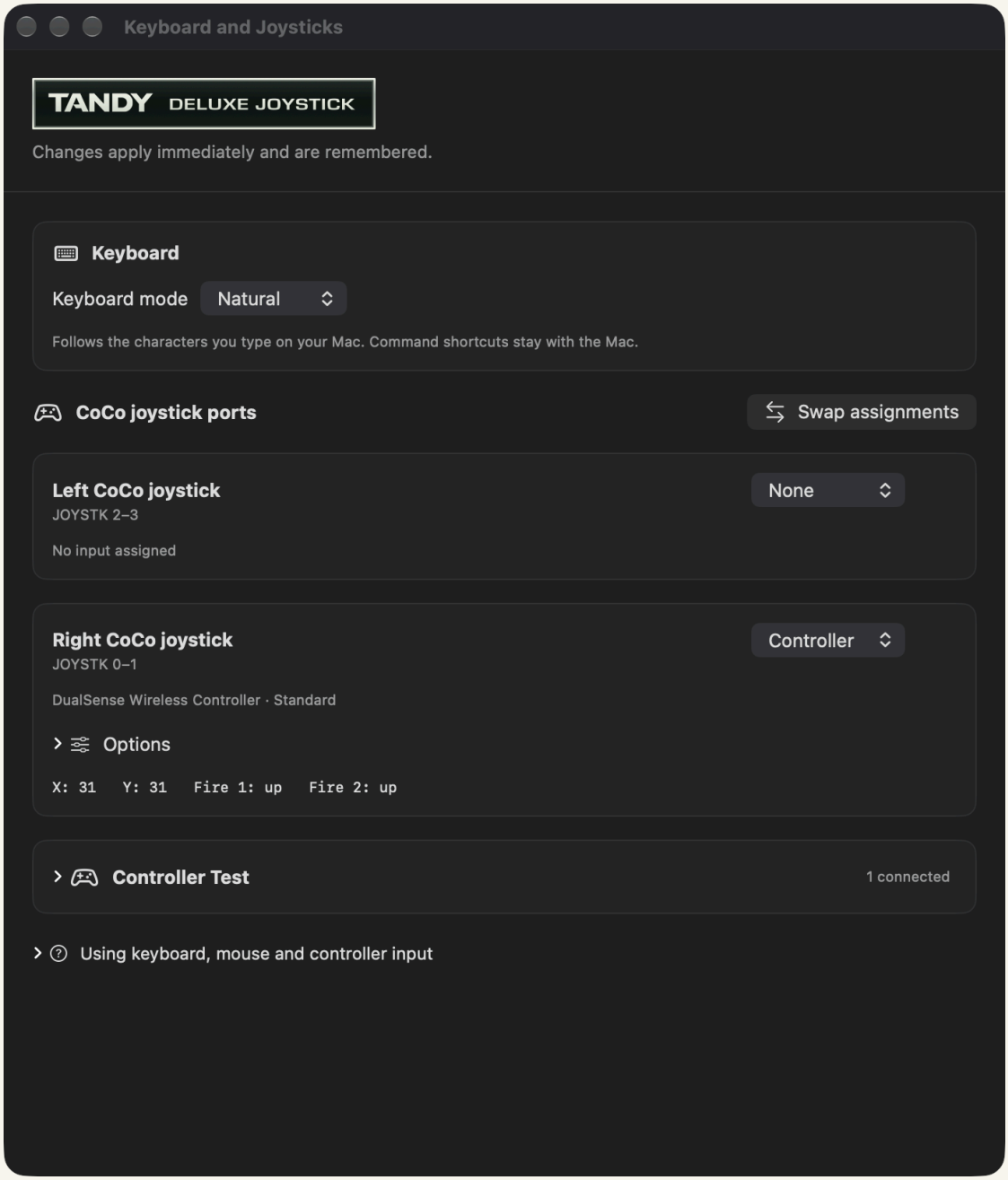


Figure 17: Keyboard and Joysticks shows separate CoCo ports. With fresh settings and one connected DualSense controller, Right selects that controller and Left remains None.

If you don't know which port a game reads, try one, then use **Swap assignments** to exchange the complete left and right setups, including their button choices. Click the live CoCo display before testing the game. Some games accept a keyboard command on the title screen and use the joystick only during play.

Follow a thumbstick and a fire button

1. Connect your USB controller or pair it in the Mac's Bluetooth settings. For a Logitech F310 on USB, set its underside switch to **D** and reconnect it. Open **Controller Test** in the input window. The connected device name should appear.

2. Move a stick and let it return to center. The *Left* and *Right* readings in Controller Test are the controller's two sticks. Watch one pair change and settle near zero. Press the physical button you want for firing; note which number appears under **Buttons**. This preview works before you assign the device to a CoCo port.
3. In the chosen CoCo port card select **Controller**. Expand **Options** and pick the device by name. The left stick is the usual starting point; choose **Use right stick** if you prefer. Set **Fire 1** to the number you just saw. If the software supports a second button, repeat for **Fire 2**.
4. Look at the *port card's* X, Y and Fire readings. Display focus may be needed for delivery to the guest. Centered values are around **31**. Moving left then right should take X toward **0** and **63**; Fire 1 goes from **up** to **down** while held. This check follows the movement all the way to the configured port. Some games use only one fire button.
5. If a resting stick jitters around the center, increase **Dead zone** a little. This ignores a small area around center. Too large a dead zone means you must push farther before movement begins. Test a small change, release the stick, and bring focus back to the game.

If the controller disappears, the assigned row can say **Remembered controller — disconnected**. The guest receives a centered stick with released buttons. Reconnect the device and check the readings again. A second device won't silently take over the missing one.

Play with a keyboard or mouse

With **Keyboard** chosen for a port, expand **Options** → **Keys**. The app offers the following layouts. The final two keys are Fire 1 and Fire 2, respectively.

Layout	Up, down, left, right	Fire buttons
Arrows	Arrow keys in those directions	Z, X
Letter keys	W, S, A, D	F, G
Numeric keypad	Keypad 8, 2, 4, 6	Keypad 0, decimal

Focus the CoCo display. Hold a direction for full deflection; release it to center. Those keys are now serving as a joystick. If typing Z at the BASIC prompt seems to do nothing, check *both* port assignments before changing the keyboard mode. Select **None** or a different source for the port to give its keys back to ordinary guest typing. If mapping both ports to the keyboard, choose different groups of keys.

With **Mouse** chosen for a port, click the CoCo picture and move over its area. The pointer's position sets the axes. Left is low X and the bottom is high Y. A left click is Fire 1 and a right click is Fire 2. Leaving the display centers the input and releases the buttons. The pointer still operates the Mac controls when you move back to them.

For a game-independent check, Point the right-port scanner uses the tested RADAR example. Hold left, center or right and tap A. The CoCo reports **WEST SIGNAL**, **CENTER SIGNAL** or **EAST SIGNAL**. Use the arrow-key layout for this exercise so A stays available to the program. To see left and right readings side by side, use Find the joystick port a game reads.

Match the guest's CoCo interface

Options → **CoCo interface** tells the emulated software which joystick interface to read. Start with **Standard** for games. **Software high resolution**, **Tandy high resolution** and **CoCoMax** are for programs that expect those ways of measuring input. Choose a special interface when the program's instructions call for it. Pairing a modern gamepad over Bluetooth, by itself, gives you no reason to change this setting.

If the character on screen still stands still, check in order. A blank **Controller Test** suggests a connection problem. A moving device preview but a still port preview points to the selected source, device or stick. A moving port preview with no response in the game points to focus, the chosen port, the game's start command or its expected CoCo interface. Restore the previous source and interface when you finish experimenting.

Keep a setup you like

Chapter 5 of 10

Suppose you lend a friend your CoCo desk. You leave a note saying "512 KB, disk controller in slot 4, practice disk in the first drive." That note is much like a VCC Plus profile. Your friend can put the desk back together. To read your program, they also need the disk you wrote it on.

Pick the machine that knows your disk

Open **Machine → Configuration Profiles...** or choose **Profiles** in the main window. Select a name first. You can read about the hardware, drives and startup instructions while the current machine keeps its place. **Load...** is the separate action that replaces the setup and clears RAM.

System profile	Where it starts	Which file or device answers?
Disk BASIC	Disk Extended Color BASIC	The first floppy is FD0. The recorder holds a separate starter cassette.
Hard Drive BASIC	RGB-DOS at a BASIC prompt	Drive 0 starts on the virtual hard disk in HD0. FD0 is also connected.
CoCo SDC	SDC-DOS at a BASIC prompt	The Mac folder stands in for an SD card. Its startup configuration mounts HELLO.DSK in SDC drive 0.
NitrOS-9 L2 V3.3.0	Boots its operating system from HD0	The system disk supplies /dd. HD1 uses the same working image that Hard Drive BASIC puts in HD0; OS-9 sees its OS-9 partition.

For the actual guest commands, try Which drive zero?. The info button on the emulator brings back instructions for the active setup. In particular, the hard-disk and SDC guides explain names that look surprisingly like ordinary floppy numbers.

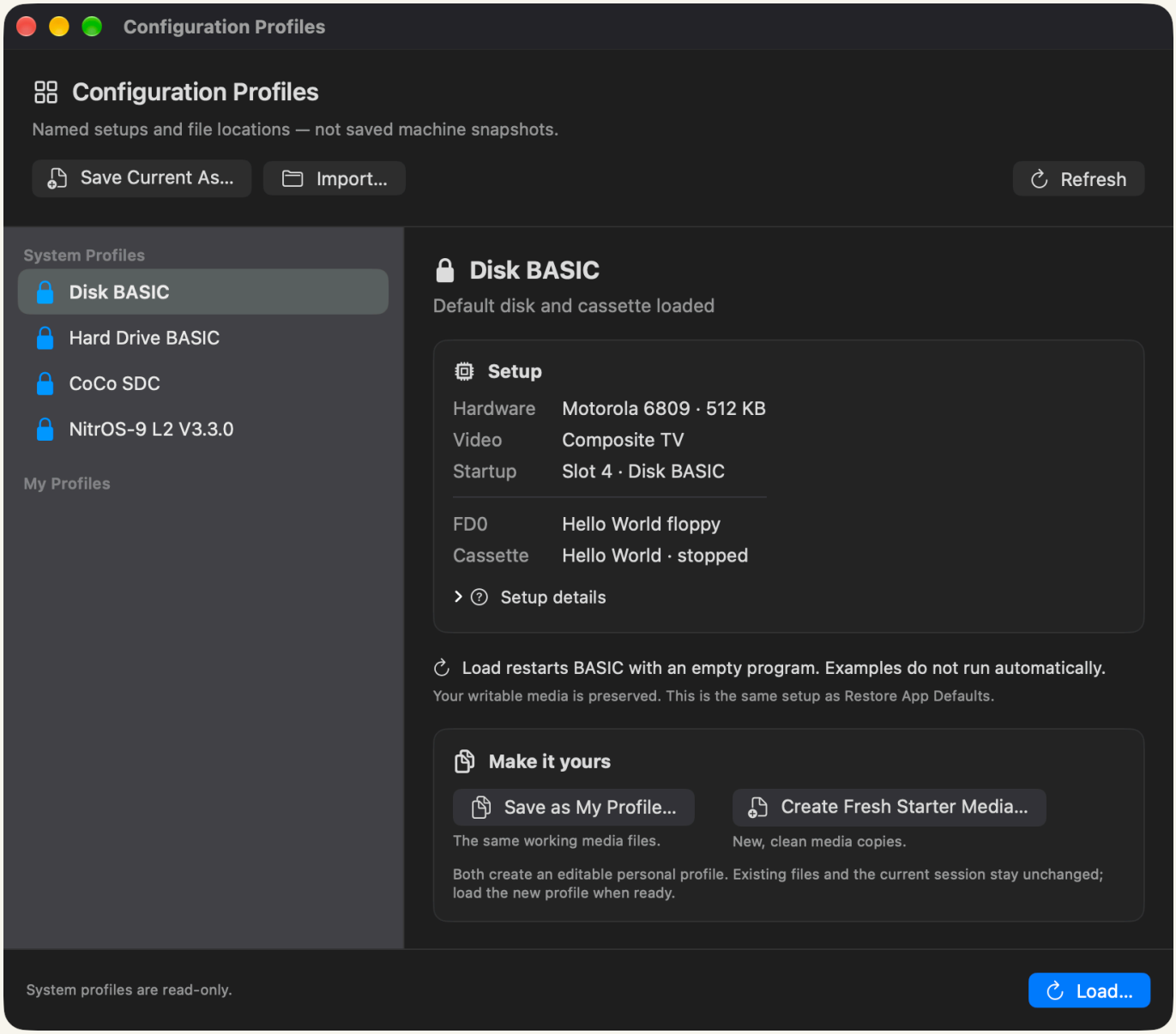


Figure 18: Select a system profile to read its hardware and media assignments before choosing Load.

Make a personal setup

Use **Save Current As...** to give the equipment on your desk a name. It appears under **My Profiles**. A personal profile remembers the processor and memory, performance settings, video output and sound, keyboard and joystick configuration, cartridge arrangement and the files assigned to devices.

If you want to start from one of the four locked system profiles instead, select it and choose how to make your personal copy:

- **Save as My Profile...** uses the same working media files as that system profile. A guest save on a shared disk can be seen the next time either profile mounts that disk.
- **Create Fresh Starter Media...** makes separate clean working copies. Use this for a new experiment if you want to leave your familiar disks in their present condition.

Creating either copy leaves you in your present session. Select the new name and choose **Load...** when you are ready to use it. Loading **Disk BASIC** restores the same standard setup as **Restore App Defaults...**, using your existing writable starter media. Choose fresh starter media when you want a clean starting disk.

Where is my work?

Where you put something	What brings it back
A source document saved in a Mac editor	Open that Mac document. A hardware restart can clear the guest copy while your saved source file stays on the Mac.
A numbered BASIC line entered at the green screen	It lives in the CoCo's RAM. Save it to a disk or tape, or read it into BASIC Development and save the Mac source, before a cold restart or profile load.
A program saved on an automatically saved, writable floppy	Mount the same Mac disk image and load the named file inside BASIC. Other profiles using that image can see later writes.
A temporary disk edit or a new cassette recording	Use **Save As…** for the disk or **Export Tape…** for the recording before leaving the setup. Look at the device's status for an unsaved change.
A drive assignment saved in a profile	Loading the profile reconnects the assigned file. Its contents come from that file as it exists now.

Try a prediction. Begin in **Disk BASIC**, with **512 KB** on the nameplate, at an empty BASIC prompt. Use a disposable, initialized FD0 with **Saves automatically** and room for a new file called PORTNOTE. If the name already exists, use a fresh practice disk.

1. Click the green screen and enter the first line below. Press Return. Enter the SAVE command and wait for **OK** before entering the remaining two lines.

```
10 PRINT "SAFE ON DISK"
SAVE "PORTNOTE"
20 PRINT "RAM ONLY"
LIST
```

2. Both numbered lines appear. Choose **Machine → Configuration Profiles… → Save Current As…**, enter **Home Port Setup**, then choose **Save**. Close Profiles. How many lines would you expect to get back if you loaded the profile now?
3. Follow Change memory and return to change RAM, load Home Port Setup and open the program you saved. You can also stop here. Until you restart, the working program still has both lines.

For the same distinction in OS-9, recover a log from a saved working image.

Manage the saved entry

Last loaded is a name to orient yourself. If you change equipment afterwards, select the personal profile and use **Actions → Update from Current…** to replace its stored configuration and file assignments. **Rename…** changes the name; **Delete…** removes that entry while keeping your ROMs, media files and current session.

For another Mac, choose **Actions → Export…** and take the resulting .vccprofile file with you. Bring the media files and ROMs you use as well. The exported profile contains local paths and controller identifiers. On the other Mac, choose **Import…**, name the new personal entry, then inspect its file references. Use **Replace** beside a moved file or card folder, or reconnect the original volume and use **Refresh**. Load the corrected profile to apply it. If a controller is unavailable, check its assignment in **Input → Keyboard and Joysticks…**.

Change the machine

Chapter 6 of 10

The emulator is halfway through a game. You want a larger picture, a faster processor or more memory. Which of those changes will take you back to the game's beginning? VCC Plus can change how it draws a running game immediately. It can also act as though you took the CoCo apart and installed different hardware. The wording of the Apply button tells you when you're about to do the latter.

Pause, reset or restart?

Choose	What the CoCo does	When to use it
Pause, then **Run**	Stops and resumes execution at the same point.	Read a fast-moving screen or leave a half-finished game where it is.
Restart → Soft Reset – Retain RAM	Starts the reset sequence while retaining the bytes in RAM. The ROM or running software decides what to do after reset.	Try the equivalent of pressing the CoCo's reset button. Save important work first.
Restart → Cold Restart – Clear RAM…	Clears RAM and starts the selected setup again after confirmation.	Start over on this machine, with its existing media.
Configuration Profiles… → Load…	Restarts with the saved machine and file assignments.	Change to different hardware or return to a setup you saved.

For the supplied NitrOS-9 system disk, see Go through the archive's menu for the extra step after Cold Restart. If the window disappeared, use **File → Show Emulator**. If the display got small, choose the presentation or resize the window. Both actions let the running program carry on.

Change memory and return

Continue the PORTNOTE experiment with two lines in RAM, one line saved to FD0 and a personal profile named **Home Port Setup**. If you're reading this section on its own, save your current BASIC program before trying a hardware change.

1. Open **Machine → Machine Settings…** (Command-comma). Under **Hardware**, select **2 MB** for **Memory**. Compare the pending configuration with the existing **512 KB** nameplate. **Revert** lets you abandon the edit here.
2. Choose **Apply and Restart…** and read the confirmation. Choose **Apply and Restart**, then close Machine Settings. The nameplate shows **2 MB**. If the machine was running, it runs again; if it was paused, choose **Run**. When BASIC is ready, enter `LIST`. Both numbered lines have gone.
3. Open **Machine → Configuration Profiles…**. Select **Home Port Setup** under **My Profiles**, choose **Load…**, then **Load Profile**. The restored machine is paused and the nameplate reads **512 KB**. In **Media → Floppy Disks…**, check that FD0 shows your practice filename and **Saves automatically**.
4. Choose **Run**. At the BASIC prompt, try `LIST`. The program area is empty. Now enter `LOAD "PORTNOTE"` and wait for **OK**. Enter `LIST`, then `RUN`, pressing Return after each. Only line 10 appears and the message **SAFE ON DISK** prints.

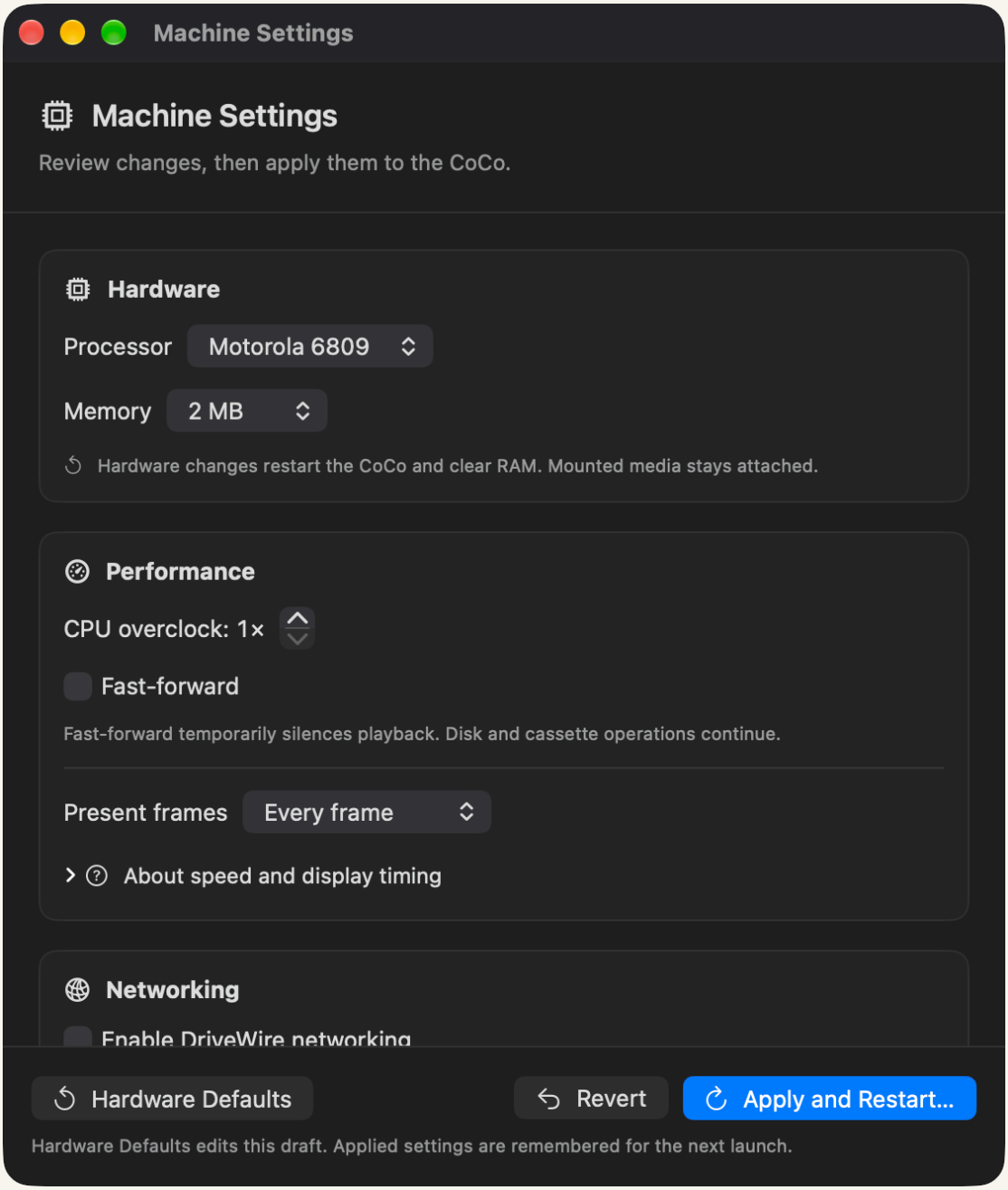


Figure 19: The Memory picker shows 2 MB. Apply and Restart is ready; the change is still a draft.

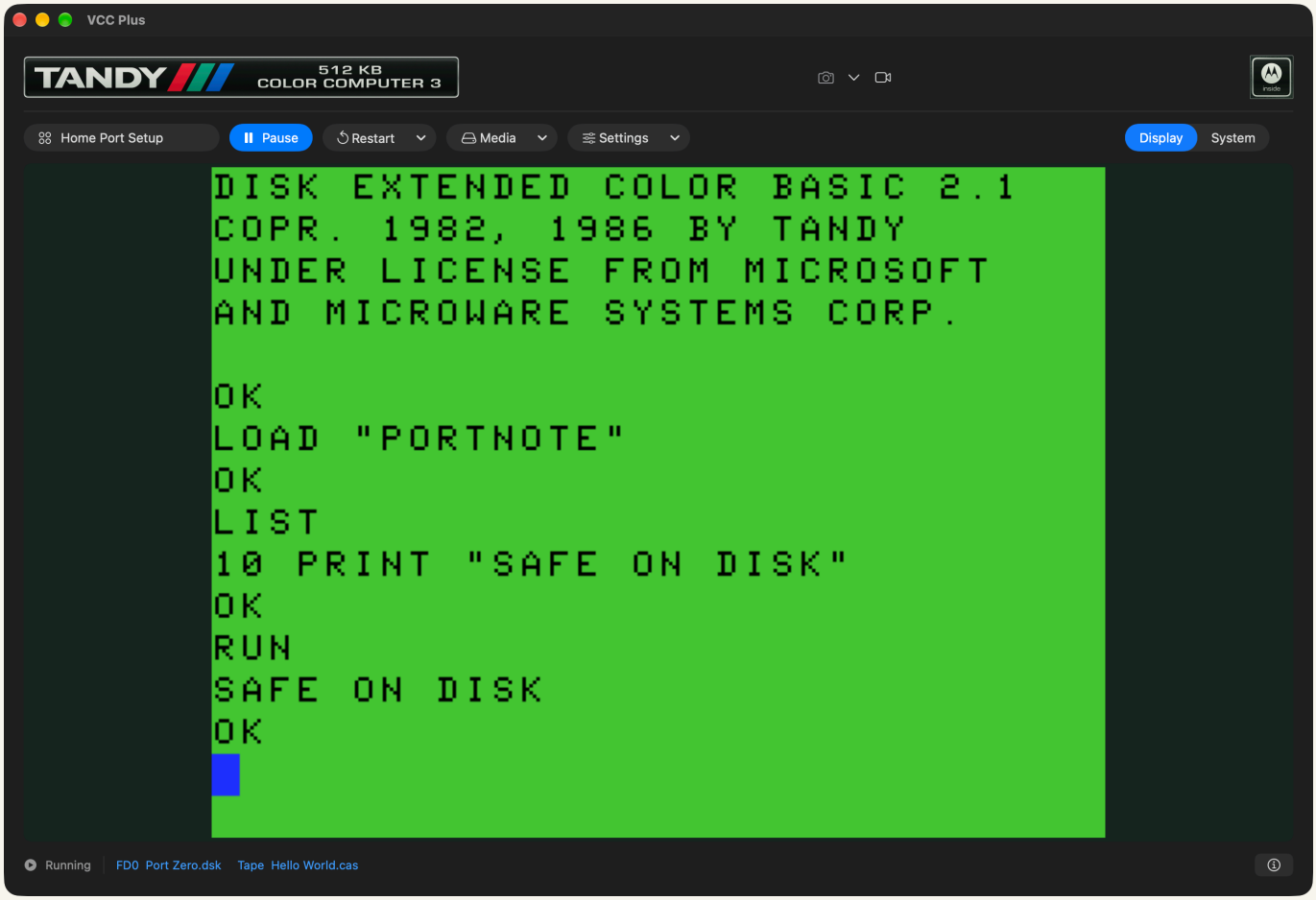


Figure 20: The 512 KB nameplate is back. After loading PORTNOTE from FD0, LIST shows a single line and RUN prints SAFE ON DISK.

You can now account for both lines. Line 10 went onto a disk before you changed the machine. Line 20 was typed after the save. The profile brought the right disk back to FD0. BASIC's LOAD brought the saved line back to RAM.

Choose the processor, memory and ROM

Motorola 6809 is the usual starting point. Choose **Hitachi 6309** when software calls for its additional instructions. The memory choices are **128 KB**, **512 KB**, **2 MB** and **8 MB**. Follow the title's requirements. For example, the Double Dragon Arcade Port on CoCo SDC calls for 2 MB. Extra RAM alone cannot make software written for the wrong processor run. Both processor and memory changes require a cold restart on a loaded machine.

A **ROM** holds code the CoCo can read at startup. **Settings → Open System ROM...** in the emulator toolbar chooses a 32 KiB CoCo 3 system ROM. The **Disk ROM** in the floppy controller is separate: it supplies disk commands. It takes an 8 or 16 KiB disk ROM. To return just one of those choices to the installed version, use **Restore Bundled System ROM...** or **Restore Bundled Disk ROM...** under Settings. Save guest work before changing ROMs.

Hardware Defaults prepares the usual 6809/512 KB draft. **Revert** puts the current applied settings back into the form. **Apply** remembers the settings for the next launch. To put the *complete* bundled Disk BASIC setup back, including its starter media assignments, use **Recovery → Restore App Defaults...**. **Settings → Run Diagnostic** starts a diagnostic preview. Return to your saved profile or to Disk BASIC when you want its normal startup screen again.

Speed, frames and sound

Performance has three controls. Change one at a time so you can tell what made a difference.

Control	What to expect
CPU overclock	**1×** is normal. Up to **16×** speeds the CPU when the CoCo program selects double-speed mode. Device and video timing keep their usual pace.
Fast-forward	Runs the whole machine ahead of real time as fast as the Mac permits. Playback is silent while this is on. Disk and cassette operations continue. The Machine menu also offers this switch.
Present frames	Every frame is the smoothest choice. Presenting every second through sixth frame reduces how often a picture reaches the Mac display. The CoCo still processes the intervening frames and its sound and devices continue.

Use **Apply** for edits in the Performance form. If a game suddenly becomes silent or races, check **Fast-forward** first. Picture size, sharpness and scanlines belong in **Display → Display Settings...**.

Let OS-9 connect

Enable DriveWire networking lets compatible OS-9 programs open outbound TCP connections through the FD-502's Becker port. The supplied NitrOS-9 setup has the guest drivers ready and defaults to networking on. Your explicit choice in Machine Settings applies across profiles. Turning it off closes active connections immediately.

A connection to the Mac shell gives a guest program the ability to run commands as your Mac user, starting in your home folder. Use it with a guest you trust. From the supplied OS-9 shell, `telnet macshell 2323` requests a session. The prompt on the Mac offers **Deny** or **Allow**. For the recognized NitrOS-9 setup you can also remember permission for programs running with that setup. **Start Mac Shell Session** lets you prepare one guest connection from the Mac. Ordinary internet access and permission to use this shell are separate choices.

Type `exit` at the Mac prompt to return to OS-9. **Stop Mac Shell Session** ends access from the Mac side. Use **Forget Mac Shell Permission** to remove a remembered grant. Read the prompt before typing: a command entered at the Mac shell can change real files on your Mac. See Ask which computer answered for the short host visit and return.

Find something to run

Chapter 7 of 10

Suppose you remember a green cover and a name, **Color File II**. You can look inside its manual before you decide whether to install the program. Or perhaps you have no title in mind and want a game for a CoCo 3. Both searches start at the wooden shelf in System view. **Media → Software Library…** opens the same window from either presentation.

Start with a curated game

The Library opens in **Getting Started**, a guided selection including **Bone Quest**, **Capital Quest** and **Quadrapassel**. Their **VCC Plus-exclusive** badge identifies this collection; they can also run on compatible real CoCo hardware. Their author and project links appear in the title details. Other Library titles retain their Color Computer Archive source credits.

On later visits the Library resumes your last browsing context: category, search, filters, view, selected title/edition and useful list position. It does not replay Play or change the CoCo's current setup. If the remembered view looks narrower than you expected, inspect the visible search and filters before starting over.

- **Bone Quest** teaches anatomy through a 28-structure identification quiz, with multiple choice and typed answers. Its setup uses **512 KB** and RGB video. Like Capital Quest, it appears under both **Education** and **Games**.
- **Capital Quest** quizzes you on all 50 U.S. states and capitals, using multiple choice or typed answers. Its setup uses **512 KB** and RGB video. It appears under both **Education** and **Games**, with Education as its primary category.
- **Quadrapassel** is a falling-block puzzle game. Its setup uses **128 KB** and RGB video. Read its manual for the keyboard, joystick and scoring choices.

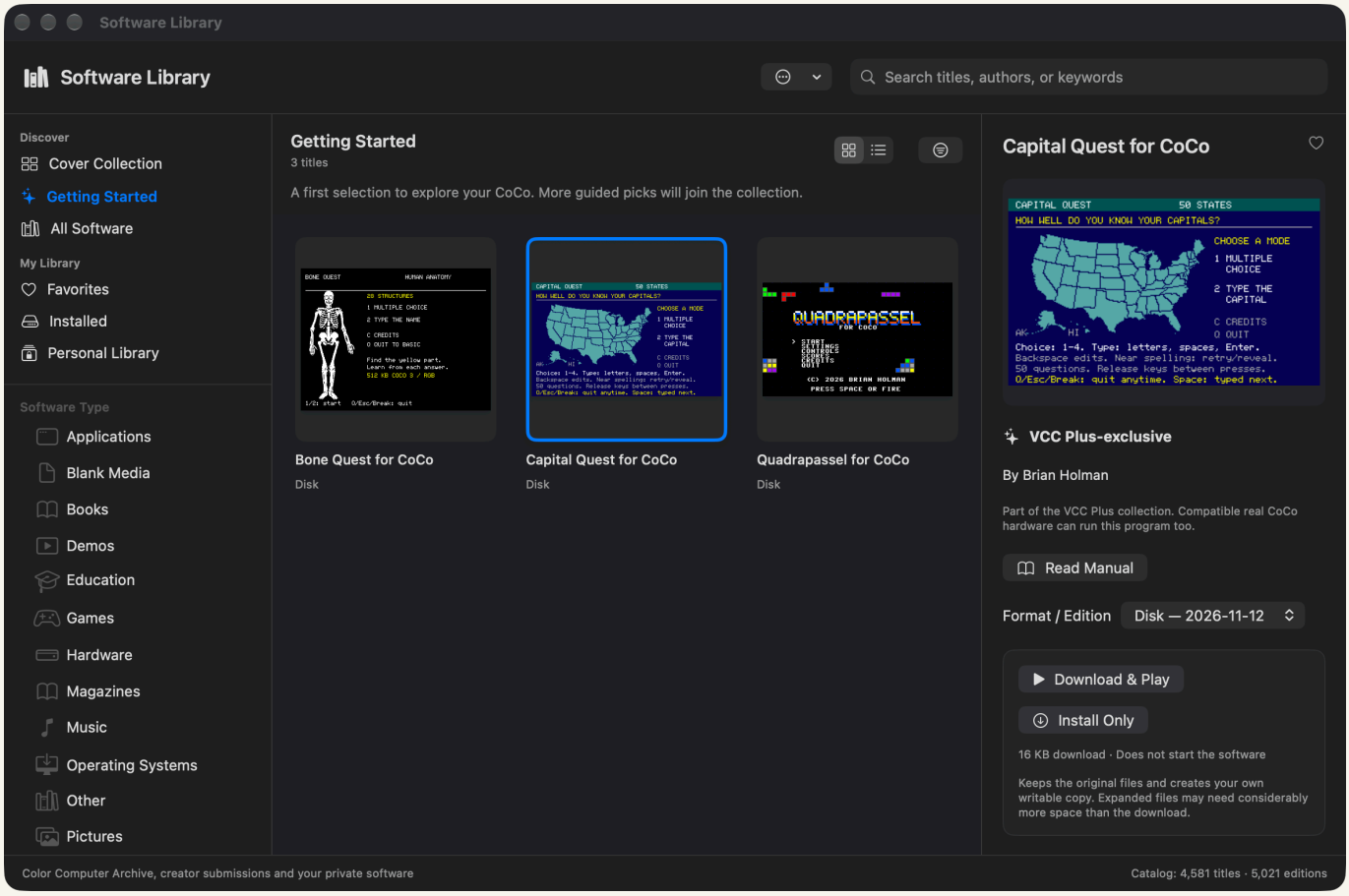


Figure 21: Getting Started shows Bone Quest, Capital Quest and Quadrapassel with their title screens. Capital Quest's details identify its author and VCC Plus collection, with Read Manual and Download & Play available.

Choose **Download & Play**, or **Play** after installation. The Library applies that title's verified setup, performs a **Cold Restart** and types **DOS** automatically. You do not need to choose a generic setup or enter the loading command yourself. Each title's specified requirements take precedence over Library defaults.

Save any guest program you want to keep first. If the current machine contains a program, an unfamiliar running setup or a RAM disk, review **Restart & Play** before continuing. Unsaved temporary media and recordings must be saved or exported before the switch. **Cancel** keeps the current session. Installed working disks retain their saved files; playing again does not replace them with the original download.

Match the game's manual to the disk edition you are playing. Capital Quest's **November 12, 2026 edition** adds **Q**, **Esc** and CoCo **Break** to quit from every screen. In typed mode, **Enter with a blank answer** reveals it as a miss; **V** reveals from spelling help. The edition chooser lets you install the new edition separately while keeping an earlier working disk and its saves. An explicit edition choice remains selected after a catalog update. **Read Manual** uses the selected edition's verified cached PDF; **Online Manual** describes the website's current edition. Opening either manual does not replace your working disk. In VCC Plus, the Mac's **Esc** key sends CoCo Break.

Find a title

1. Search for **Color File II** in the header. Typing opens **Search Results** across catalog and Personal Library titles, authors and keywords, even when you began in Getting Started or a different category. You can then apply filters to those results.
2. Select the title. The center of the window lists matches; the panel on the right describes your selection. Widen the window if you want more room. **Shelf** shows covers face out. **Table** makes it easier to scan names and formats.

3. Choose **Read Manual** if you want to recognize the title by its cover. You can also choose the heart beside the title to find it again under **Favorites**.

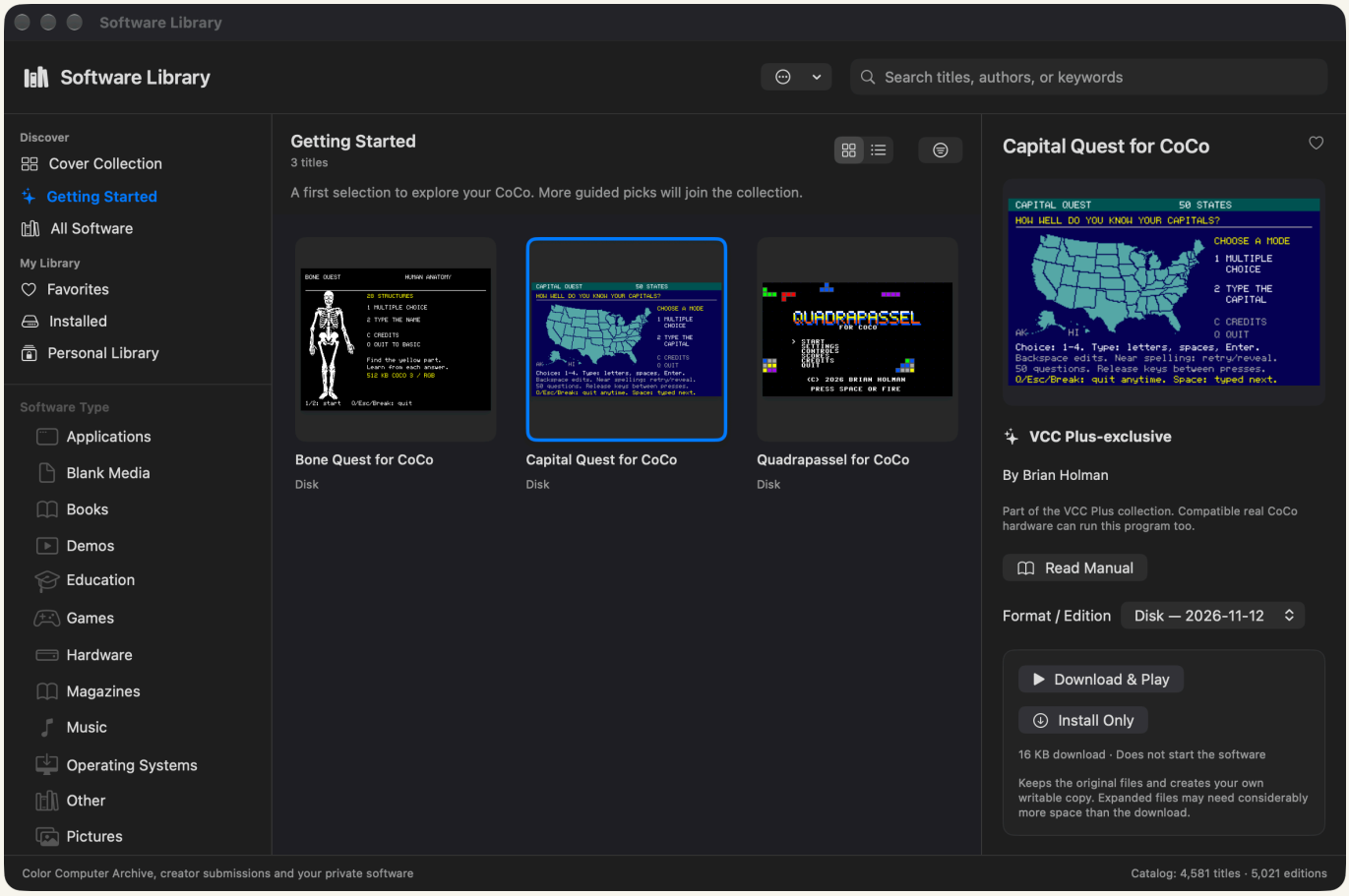


Figure 22: The Software Library has a sidebar for choosing where to browse, a list of matching titles, and details for the selected item.

Cover Collection is the illustrated selection. **Getting Started** offers a smaller place to begin. **Installed** lists editions you have installed on this Mac. To browse by subject, choose a **Software Type** such as Games, Education or Music. A title can belong to more than one category without appearing twice in a category's list. Personal matches appear first alphabetically with a **Personal** badge, followed by curated catalog titles alphabetically and then the other matching catalog titles alphabetically. If you choose modification date or download size as the sort, that choice applies to the remaining titles while the curated group stays alphabetical. To browse the way the Color Computer Archive organizes its collection, expand **Archive Folders**.

If a familiar title disappears, check the search box as well as the sidebar. For example, switching from Games to Education keeps the words you typed. Clear the search to see everything in Education. Typing a new search from a browse category opens the global results and clears that old scope and its filters. Filters you deliberately choose within Search Results remain as you refine the words. Clearing a global search returns to **All Software**. **Browse All Titles** on the empty-results screen clears the search and filters.

Choose a version that fits

Use **Filter and sort** to narrow the titles. **Software Type** answers "What is it for?" **Format** answers "What does it come on?" A game might come on a cartridge, cassette or disk. **System** helps find titles for a particular model. **Environment** distinguishes software intended for systems such as Disk BASIC or OS-9. Use **Setup** when you care how much of the startup the Library can do for you. Sort by title, modification date or download size.

Under **Advanced Filters**, **Processor** can narrow a search to software requiring a Hitachi 6309. **Hardware** and **Use** let you distinguish a requirement from support for an accessory; for instance, a game can offer joystick controls as an option. **Display** includes semigraphics modes. Some catalog entries have little descriptive information, so adding filters can hide a title that would work. When a release carries the **6847T1** badge, that is an Archive note about the video chip. Look in the edition's compatibility information for any VCC Plus checks.

A title can have several editions. Choose the format and edition in its detail panel before reading the setup or pressing Play. Under **Using this software**, look for the recommended machine, loading instructions and any known controls. Read **Compatibility** separately for requirements or limitations. **View on Color Computer Archive** takes you to the collection that preserved the software.

Project-owned titles instead provide **Visit Project**, their own author and license, and a separate **Online Manual**. Those links describe their actual source rather than attributing them to the Archive.

Read a manual before installing

Read Manual opens a PDF. Turn pages with the arrows, enter a page number, open the thumbnails or choose **Fit Page** to bring a whole page back into view. **Find Next** can locate words in a searchable document. On a scan with no searchable text, use the thumbnails to spot the contents page. **Read Instructions** opens a text document, often the readme the author supplied with the download. For the SDC edition of **Double Dragon Arcade Port**, check the controls in the title panel as well as the readme before trying to get past its arcade title screen.

Reopening the same software manual resumes its own saved reading position, including supported PDF zoom or text scroll. A different edition's manual keeps its own position. This does not change the selected working disk or rerun the software.

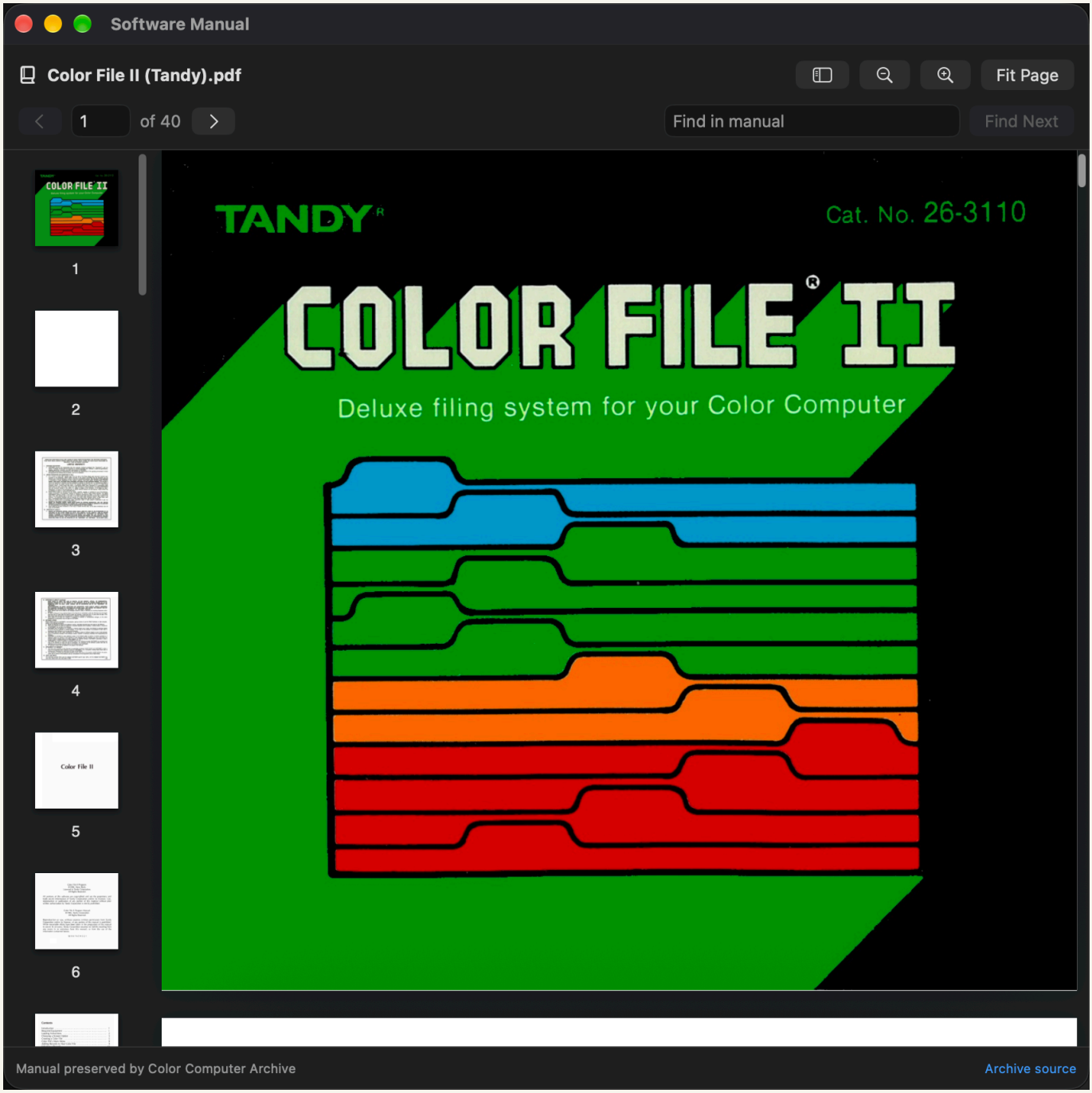


Figure 23: Color File II's original green cover in the Software Manual window. Page thumbnails are open on the left; the page field, search, zoom and Fit Page controls are above it. Cover by Tandy, preserved by the Color Computer Archive.

The catalog on your Mac lets you browse before downloading a program. Covers and software documents are fetched as needed and cached locally. Expect to need a connection the first time you open one. **Help → Manual Library...** has the bundled VCC Plus books for offline reading, including this handbook.
Looking for a magazine review, a tutorial or another edition of a manual? Open **Help → Archive Library...** and search the publication pages for the software title. Find a page in the Color Computer Archive.

Follow a guide or reference

Some titles have **Guides & References** below their manual and setup information. These are reviewed links to supplementary material in the Archive: a walkthrough, map, tutorial or useful reference. For example, Raaka-Tu's

walkthrough is separate from its main manual. A walkthrough or solution can say **Contains spoilers** so you know what you are opening.

Select the document once. Archive Library opens the actual text, image or cited publication page in its existing reader; there is no second Read button. The Software Library keeps its title/edition selection and the CoCo keeps running. A page-specific reference opens that page even if you read farther last time. A document-only reference can resume that document's own remembered position. The original Color Computer Archive credit and citation remain available there.

The links apply to the selected software edition. An entry with no reviewed association has no empty Guides & References section. A missing or changed reference reports that it is unavailable rather than opening an unrelated title. You can still search Archive Library directly for more material.

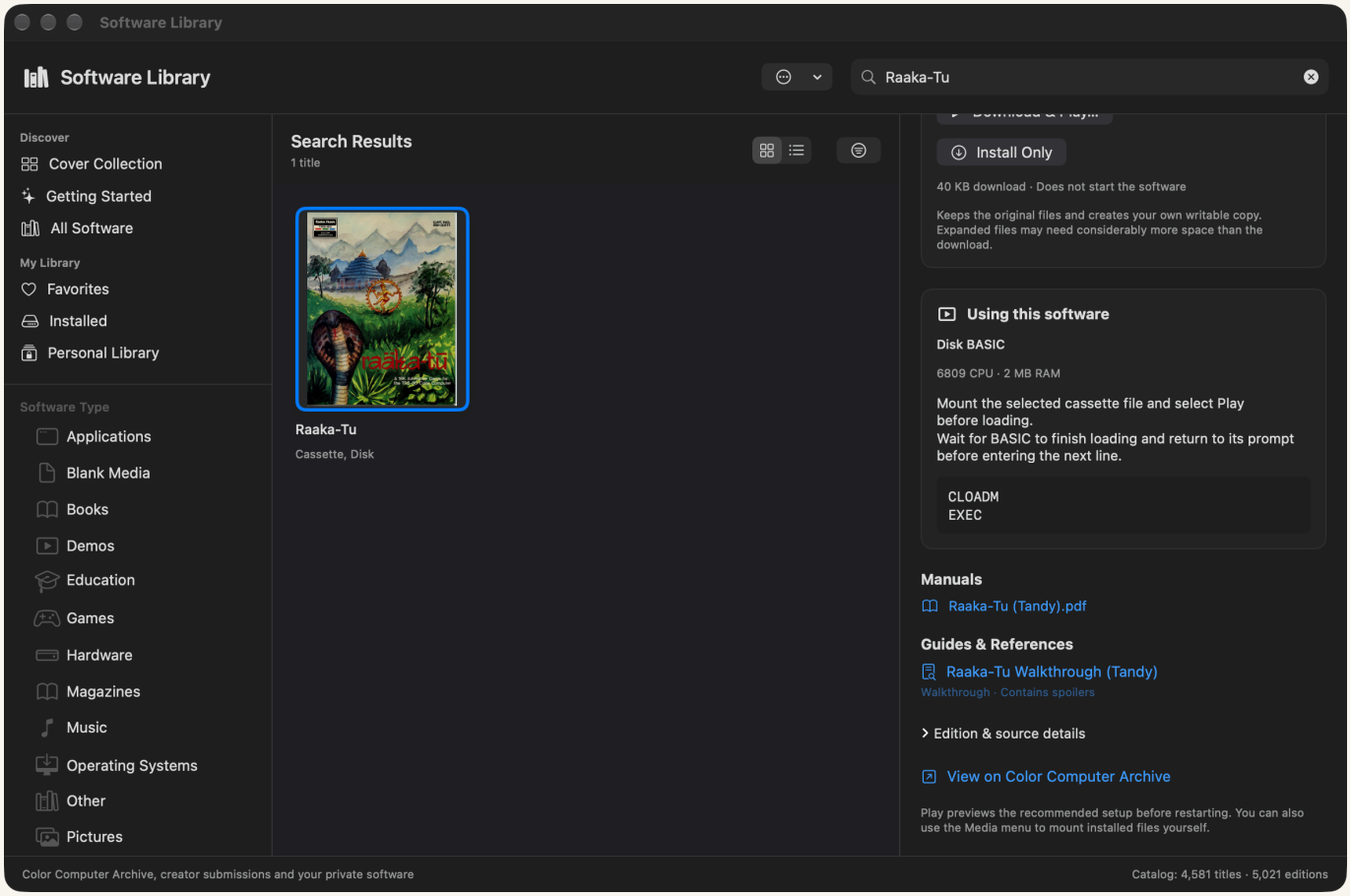


Figure 24: Raaka-Tu lists its main PDF under Manuals and a separate Raaka-Tu Walkthrough under Guides & References. The walkthrough is marked Contains spoilers. The game artwork and supporting documents are preserved by the Color Computer Archive.

Add software you own privately

Select **Personal Library** under My Library and choose **Add Software...** below the heading. This button stays available as your collection grows; an empty library also offers it beside the invitation to add your first software. From any other category, use **Library Actions**, the ellipsis icon beside the search controls, and choose **Add Personal Software...**. Select individual files, one folder or a ZIP, then choose **Review Files**. This creates a private collection on this Mac. It is useful for a purchased program, a personal project or another title you do not want to submit for public distribution.

The CoCo still has an active community of developers who sell their software. Much historical software is generously shared, but availability does not mean every creator has given permission to redistribute their work. Support the developers whose programs you enjoy, buy from their authorized stores, and respect their wishes

about copying and sharing. Personal Library gives you a convenient home for software you have purchased or otherwise obtained with permission, without adding it to the public catalog.

One example is Nickolas (Nick) Marentes, whose store offers both commercial games and freely available titles. The screenshots in this section show Brian Holman's personal copies of Nick's PAC-MAN tribute and PopStar Pilot, with their cover images credited to Nick. Brian bought software from Nick around his visit to CoCoFEST 2015. These examples illustrate organizing a private collection; the games are not included with VCC Plus.

In **Files & Details**, confirm the title and select the members you want to keep. Author/publisher, software version, category and notes are optional. Supporting manuals and images can stay with the software. Unrecognized files are not treated as runnable media, and nested archives are not expanded automatically.

Choose one standard **Category** from the dropdown; it starts at **Other**. The category supplies the fallback illustration and places the entry in the matching browse category. Existing unrecognized categories become Other.

In **Cover Image**, select an imported image, choose **Choose Image…** to browse for a local picture, or drop an image onto the preview. The preview preserves the picture's proportions. Save the entry to keep the change; Cancel leaves its previous image in place. **Remove Cover** restores the category illustration. After import, the detail panel's **Cover Image** menu and image drop area let you replace the cover directly. Pictures are copied into this Mac's private library; the source image, software files and saved games stay unchanged. Your chosen cover also appears in the matching DriveWire library through its local connection.

Launch Setup starts with **Use reviewed launch settings** off. You can leave the entry in Manual Setup, or review its machine setup, processor, RAM, video, media assignments and optional keyboard loading commands. Assign a multi-disk set to the intended FD0–FD3 or HD0–HD1 roles; a cassette or ROM cartridge has its own role. Commands are CoCo keyboard input, not Mac shell commands. Leave a sequence blank when you do not know it rather than guessing. A single imported floppy defaults to FD0. Multi-disk sets need explicit assignments; existing saved drive choices are preserved.

The command editor accepts multiple lines: press Return to start the next line. For a disk BASIC program, enter the loading and running commands separately:

```
LOAD "PROGRAM"
RUN
```

Replace `PROGRAM` with the program's disk filename. Ordinary `LOAD` discards the rest of its command line, so `LOAD "PROGRAM":RUN` is not an automatic startup sequence. Once machine settings and media assignments are reviewed and saved, the entry offers **Play…** even with no keyboard commands. Play opens the configured CoCo with its media ready; supported saved commands are entered automatically, or you can load the program yourself. **Set Up…** remains for entries without reviewed media assignments. An explanation appears in the editor when entered commands need manual entry. Each supported line uses ordinary emulated keyboard input and the normal readiness checks.

Choose **Add to Personal Library**. Import copies the selected files without changing the purchased source folder, starting software or reconfiguring the guest. An identical set offers **Open Existing** or **Add Separate Copy**. The originals, writable working media and supporting assets are retained separately.

Find these entries under **Personal Library** in My Library. They say **Personal Library — Private on This Mac**. **All Software** and the standard categories include both personal and catalog entries. Typing in the top search field opens **Search Results** across both collections, even when you started in Getting Started or Personal Library. Previous category and format filters are cleared when global search begins; filters you choose within results remain visible. Clearing the search returns to All Software.

Matching personal titles appear first, alphabetically, with a lock and **Personal** badge on cards, table rows and details. Mixed shelves group Personal Library above Catalog; the catalog retains its normal curated/category sorting. Same-named personal and catalog editions stay separate. **Personal Library** itself shows only private entries, and Favorites can include either collection. A deliberate **Play** or **Set Up…** still uses the normal replacement and unsaved-media safeguards. For a stored binary, use **Show Files in Finder**, then **File → Load Binary…** to choose it explicitly.

Edit Details... changes the description and reviewed setup without rebuilding working disks or losing saved games. **Show Files in Finder** opens the writable Working folder. Go up one folder in Finder to see the retained entry and its originals/assets; back up the complete entry when you want to preserve it. **Remove from Library...** hides its listing but retains its files and saves, so no disk space is reclaimed. Use Finder for a deliberate permanent deletion only after the media is no longer in use. App and public-catalog updates preserve the private collection. Import does not upload files or submit a title for public distribution. A matching current **DriveWire for VCC Plus** can browse your personal disk entries through the local companion connection. Insert Disk makes a separate writable DriveWire copy, preserving the originals and emulator's working disk. Add, edit and remove personal listings in VCC Plus.

Favorites and selected editions are shared between the two apps. After changing your entries, cover images or choices in VCC Plus, close and reopen DriveWire's Software Library to see the latest collection. Changes to a DriveWire disk stay in its separate copy; they do not overwrite your emulator's working disk. To use those changes in VCC Plus, save that disk copy and explicitly open or import it.

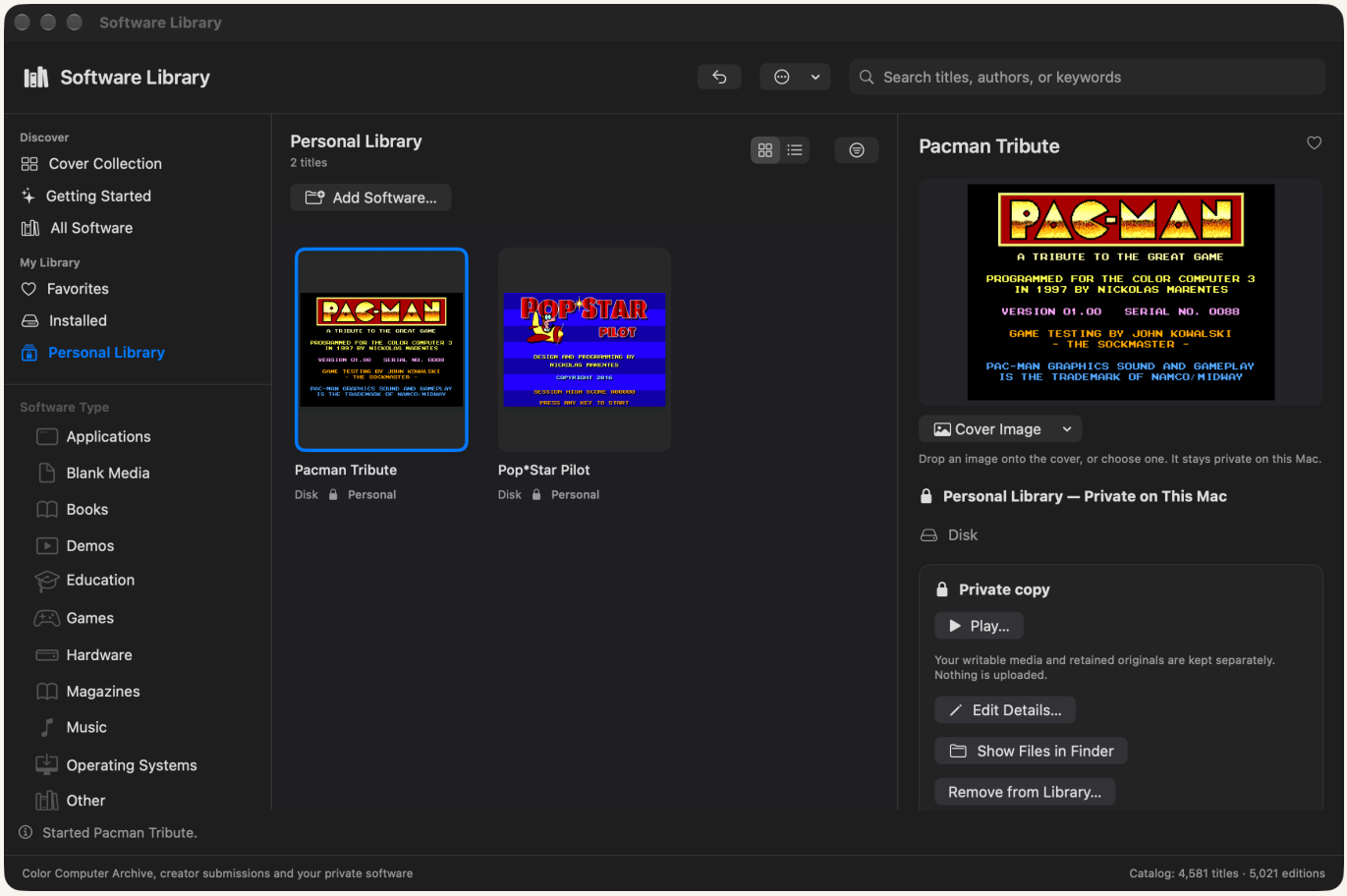


Figure 25: Personal Library keeps Brian's PAC-MAN tribute and PopStar Pilot together with custom cover images, Personal badges and a saved Play setup. Games and cover images by Nickolas Marentes; these personal copies are not bundled with VCC Plus.

Offer software or request its removal

Creators can use **Library Actions → Submit Software...** to open the website's submission form. This is a separate choice from adding a private copy. Provide the title, author, version or historical details, software package or link, contact information and any supporting manuals or artwork. Identify included third-party material and review the permission statement before submitting.

Submission grants the permissions stated in that form to **Dezota LLC, operating VCC Plus**; it does not transfer your copyright or automatically make the program open source. The website returns a reference for the review

queue. Submission, approval and inclusion in a published VCC Plus release are separate steps. Opening the form sends no software or message by itself.

The submission page links to **Request Software Removal or Withdraw a Submission** in the website feedback service. You can identify a direct submission or an Archive-sourced title and provide evidence that you hold the rights or are authorized to act. A verified rights holder can withdraw future VCC Plus distribution even if the software remains on Color Computer Archive. VCC Plus controls its own catalog and downloads, not the Archive's independently operated website. A removal request does not remotely erase users' existing copies.

For an ordinary bug report or suggestion, use **Help → Feedback...** instead. It opens the website with VCC Plus preselected, lets you review the information and submits only when you choose to send it.

Open a personal source project

Bone Quest, Capital Quest and Quadrapassel offer **Open Source...** under **Learn from the source**. Choose a new folder on your Mac. The Library downloads and verifies the published starter, prepares a writable personal copy, and opens its known project in **C Development**. A starter can specify a build configuration; Quadrapassel opens in **Release**.

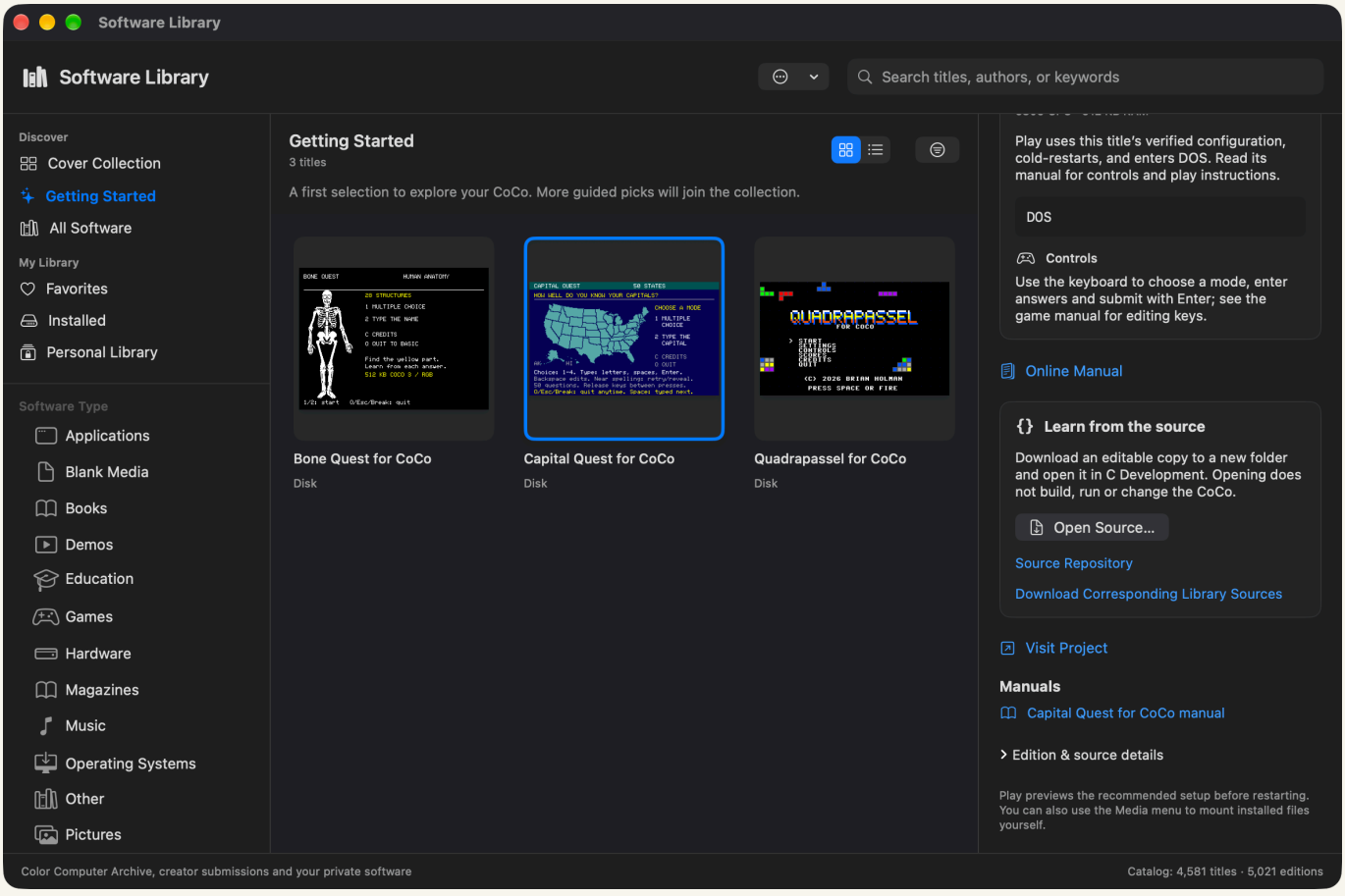


Figure 26: Capital Quest's detail panel is scrolled to its DOS startup, keyboard controls, Online Manual and Learn from the source. Open Source creates a personal copy; the repository and corresponding library sources are separate links.

Opening source does not evaluate its Makefile, build or run the program, mount a disk or restart the CoCo. When you choose **Build**, the C workspace uses the project's build recipe. Read its source and project documentation before making changes.

If another C project has unsaved edits, choose **Save All**, **Cancel** or **Discard Changes** in its review sheet. Cancel leaves the old editor and its drafts open while retaining the newly downloaded folder. If you changed projects

during the download, the Library keeps that new workspace and offers **Show Source Folder** so you can open the imported project when ready. Existing personal project folders are never overwritten or merged.

Source Repository opens the project's online history. **Download Corresponding Library Sources** is a separate download of the matching library source and notices; it is not automatically unpacked into the starter. **Read Manual** uses the verified PDF in the Library's normal download storage, while **Online Manual** opens the website reading edition.

Install, play or finish the loading yourself

- **Install Only** downloads and verifies the selected edition and prepares working files. You can keep using the current CoCo while deciding when to play. If a download fails, use **Try Again**.
- **Download & Play…**, or **Play…** for an installed Archive edition, previews the recommended setup. Read the processor, RAM, video and mounted media. A package with several starting files may ask you to **Choose Starting Media** first. Use the author's loading notes to choose the file. The verified curated game flow described above handles its setup and DOS command directly.
- **Download & Set Up…**, or **Set Up…**, prepares an edition that needs manual loading. The preview tells you what it will mount. Once the guest is ready, follow the displayed instructions, using the game's own manual to choose a program or answer a prompt.

Save any program you want to keep before confirming the restart. RAM will be cleared. You can still **Cancel** at the preview and use the installed files later. When you confirm an automatic start, watch the CoCo screen; **Stop Automatic Loading** stops further automatic input if you prefer to take over. Click the CoCo display when it is time to type or play.

Come back with your files

Return to Previous Setup… is available when the Library captured your previous configuration. It restores that hardware and those media assignments through a restart. A program that was sitting in RAM before your visit needs to be reloaded from its saved copy. See how a returning profile brings its disk home.

Under **Installed**, **Show Files in Finder** opens the working copy, including writable media that can hold a saved game. The original download is kept separately. **Remove from Library…** removes the installation record while preserving the files on this Mac. Installing again makes a fresh working copy; use Finder to locate any previous save you want to keep playing.

Put the right file in the right device

Chapter 8 of 10

A .dsk file in Finder can hold a dozen CoCo programs. Picking that file is like sliding a floppy into a drive. You still have to tell the guest which program to open. There are three names worth checking before a save: the Mac file, the device holding it, and the filename *inside* the disk.

Open a file or return to one

Choose **File → Open Media...** or drop a file from Finder onto the emulator. Check the proposed destination before continuing. A cartridge goes in a cartridge slot, a tape in the recorder and a disk in a drive. The **Media** menu takes you straight to a device. The pictures and named **Devices** controls in System view are other ways to get there.

File → Open Recent Media remembers files you opened and their destinations. If the file moved, locate it again; a remembered name supplies no copy of its contents. **Clear Recent Media** clears that menu. To start a CoCo block-format binary directly, use **File → Load Binary...**. The loader writes the program's address ranges into guest memory and starts it without a reset. Save your current program first. Canceling the binary picker leaves the emulator paused.

When opening a floppy image from Finder or Open Media, choose **Edit Disk Image** to inspect its files, or **Insert into Floppy Drive** to put it in FD0–FD3. Insertion offers **Keep current state**, **Pause**, or **Run**. Run resumes the computer after insertion succeeds; you still choose what to load or boot from the disk.

Before inserting offers **No Reset** (the default), **Soft Reset**, or **Cold Restart**. Soft Reset restarts the guest without clearing RAM, although its ROM reset routine may change RAM; Cold Restart clears the current program and memory. Save anything you need first. The chosen reset occurs after the disk is validated and replacement safeguards succeed, before insertion. The separate execution choice determines whether the machine runs or remains paused afterward. Opening the editor performs no reset.

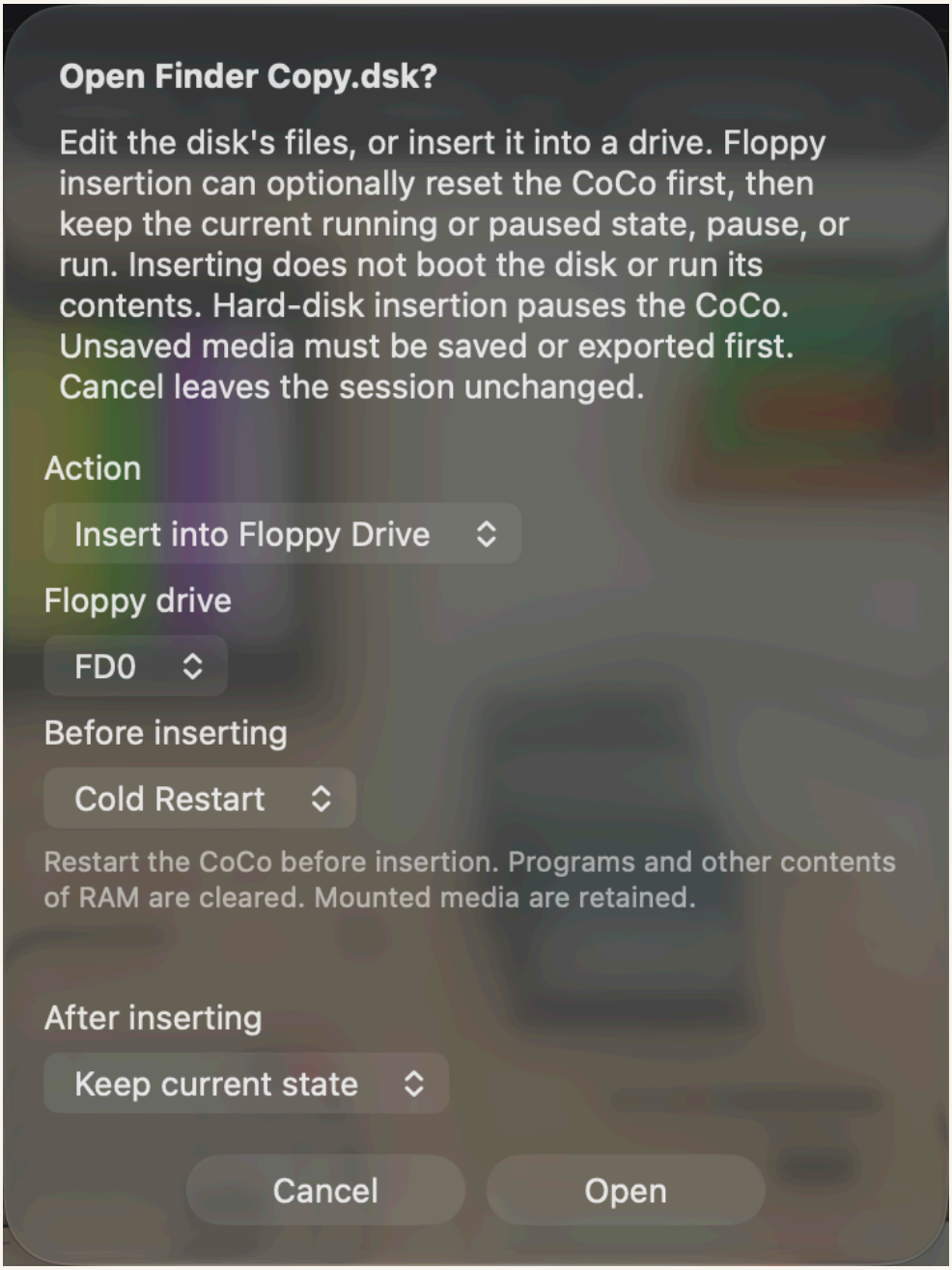


Figure 27: Opening a disk from Finder offers Edit Disk Image or insertion into a selected floppy. Before inserting selects No Reset, Soft Reset or Cold Restart; After inserting independently selects the execution state.

Keep a floppy safe

Open **Media → Floppy Disks...**. The FD-502 serves four drives, **FD0–FD3**. Read the filename and state beside the drive you intend to use. Choose **Insert...** for an empty drive or **Replace...** for an occupied one. Media and save-mode changes pause the CoCo; click **Run** when you're ready. An ordinary automatic save continues while the CoCo runs.

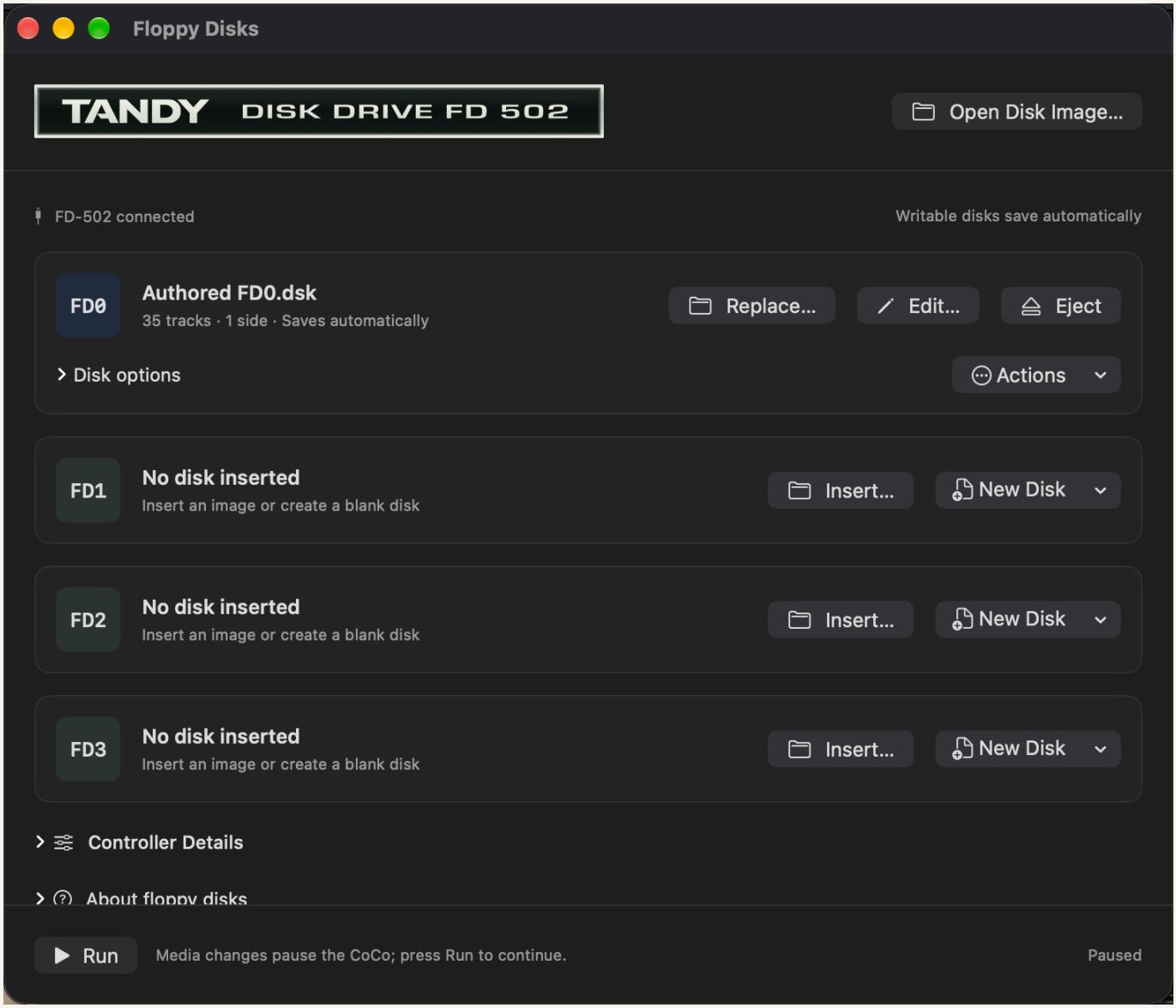


Figure 28: The floppy window identifies each drive separately. Read FD0's filename and save mode together before writing.

What you want	What to do
Save ordinary work	Mount a writable image and check for **Saves automatically**. When a guest program writes a file, VCC Plus saves the changed image to its Mac file.
Examine an original safely	Open **Disk options** and enable **Write protected**. The guest can read the image. A read-only Mac file or a protection flag in the image header can lock this switch.
Let the game write, but decide later whether to keep the result	On a writable disk, turn on **Temporary changes** before the game writes. **Save As...** keeps a copy and retains that mode. Turning the mode off can write pending changes to the assigned file.
Keep a disk after a save error	Read the affected drive's message. If you disconnected its volume, reconnect it and choose **Retry Save**. If someone edited the file elsewhere, choose **Save As...** under a different name to preserve both versions.

What you want	What to do
Deliberately leave changes behind	**Discard Changes and Eject...** gives up pending in-memory edits. Any writes that already reached the Mac file are part of that file.

An unavailable assignment and a failed write call for different actions. At startup, **Retry Remembered Disks** looks for the assigned files again. You can also choose a replacement or eject the outdated assignment. During a save failure the app keeps the changed disk in memory so you can rescue it. A read/write light reports real activity on the named drive. A warning in the drive row tells you when action is needed.

New Disk offers **Standard DSK**, **DMK Track Image** and **VDK Disk Image**. Each creates a blank, single-sided 35-track disk in memory. For a *new, empty practice disk in FD0 only*, use the Disk BASIC command `DSKINI0` to format it, then choose **Save As...** to give it a Mac filename. Formatting an existing disk destroys its directory and files. The FD-502 can also open compatible JVC and OS-9 floppy images. **Save As...** preserves the mounted format. For a ready-made, protected example, follow Put the practice disk in FD0.

For a working-copy exercise you can do entirely offline, change one copy and reopen both.

Controller Details shows the controller ROM, selected drive, track, sector and motor status. Choose a compatible disk ROM here when supplying your own. Disabling this controller ejects all four disks and cold-restarts the CoCo. Resolve pending writes first.

Edit the files inside an image

Choose **Edit...** beside an occupied **FD0–FD3** in Floppy Disks or **HD0/HD1** in Virtual Hard Disks to open the native Disk Image Editor. The machine pauses and stays locked while you edit its current disk, including temporary changes. **Open Disk Image...** in Floppy Disks opens an unmounted floppy or hard-drive image without inserting it or disturbing the CoCo. If that file is already mounted in a floppy or hard drive, including through a file alias, VCC Plus uses the mounted-disk lock. An image assigned to another device or an SD card folder must be unassigned before standalone editing.

The file table shows names, sizes, types and filesystem details. The SF Symbols toolbar imports and exports files, renames or deletes entries, creates OS-9 folders, and offers Undo/Redo. Double-click a folder to enter it, or a file to export it. Drag ordinary Mac files into the table to import them. Drag selected regular files to a Finder folder to export copies. Open the intended OS-9 folder first. Total, Used and Free show actual allocation space, including allocation rounding and filesystem overhead. Disk BASIC excludes its reserved directory track from Total.

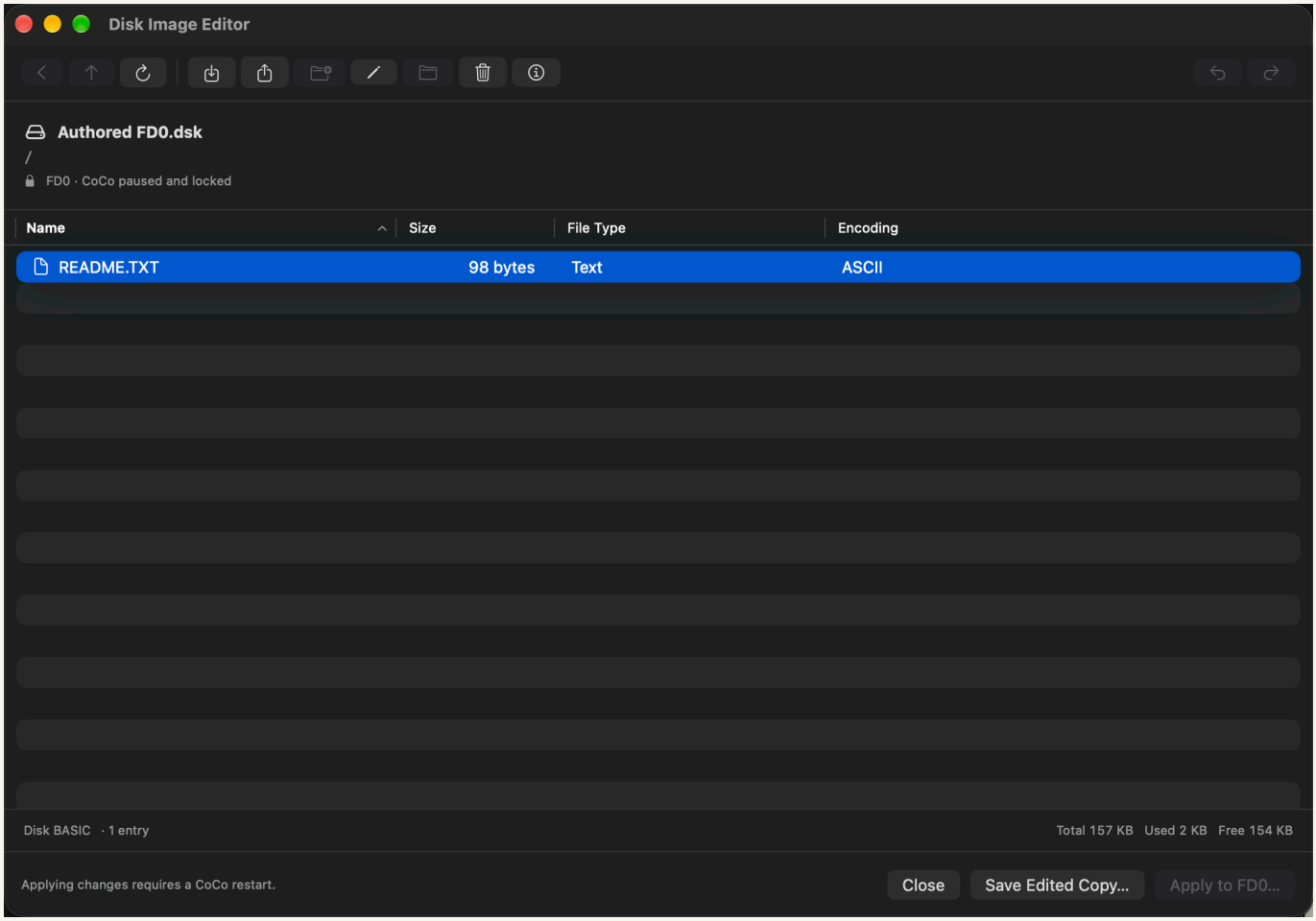


Figure 29: The Disk BASIC table shows file type and encoding, with total, used and free file space below. OS-9 folder controls are unavailable for this filesystem.

Edits stay in a private working copy. **Save Edited Copy...** keeps a separate image. **Save Changes...** updates an unmounted writable source after checking for external changes. For a mounted disk, **Apply to FDn...** or **Apply to HDn...** asks for **Restart and Apply**: the running guest program and memory will be cleared to prevent stale filesystem information. Save programs and recordings you need first. Cancel or Discard Changes leaves the original disk intact. Closing a mounted editor leaves the CoCo paused.

For an OS-9 file or folder, **Info** opens one File Information sheet for owner/public read, write and execute permissions, single access, and ownership. Group ID and User ID are each 0–255; together they represent the OS-9 owner ID shown as `group.user` in the table. File versus Directory stays fixed. Save Changes updates the working copy and supports Undo; Cancel leaves it alone. Applying or saving the image is still a separate action.

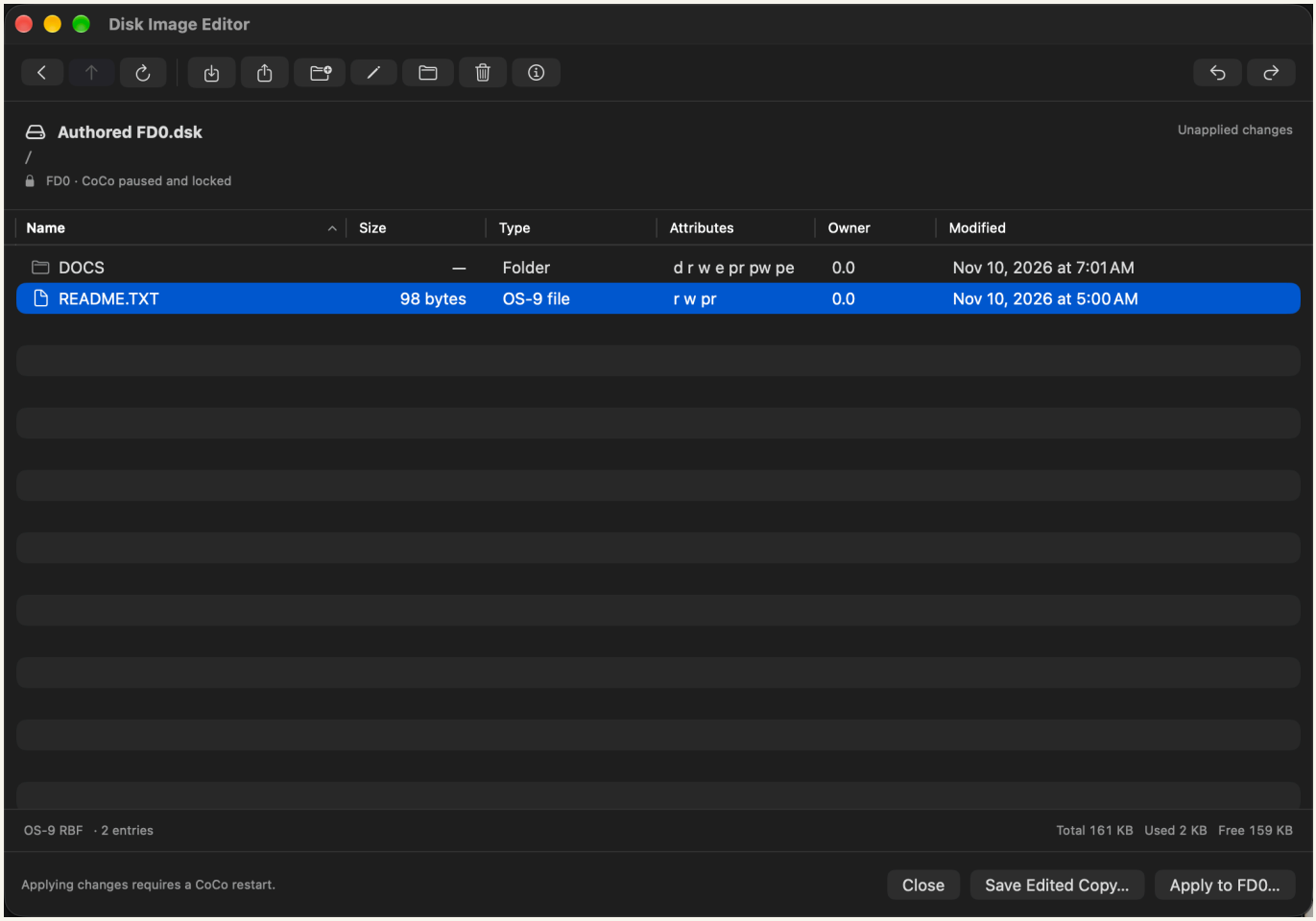


Figure 30: The OS-9 view adds attributes, group.user ownership and modification dates. Folder actions become available for this filesystem.

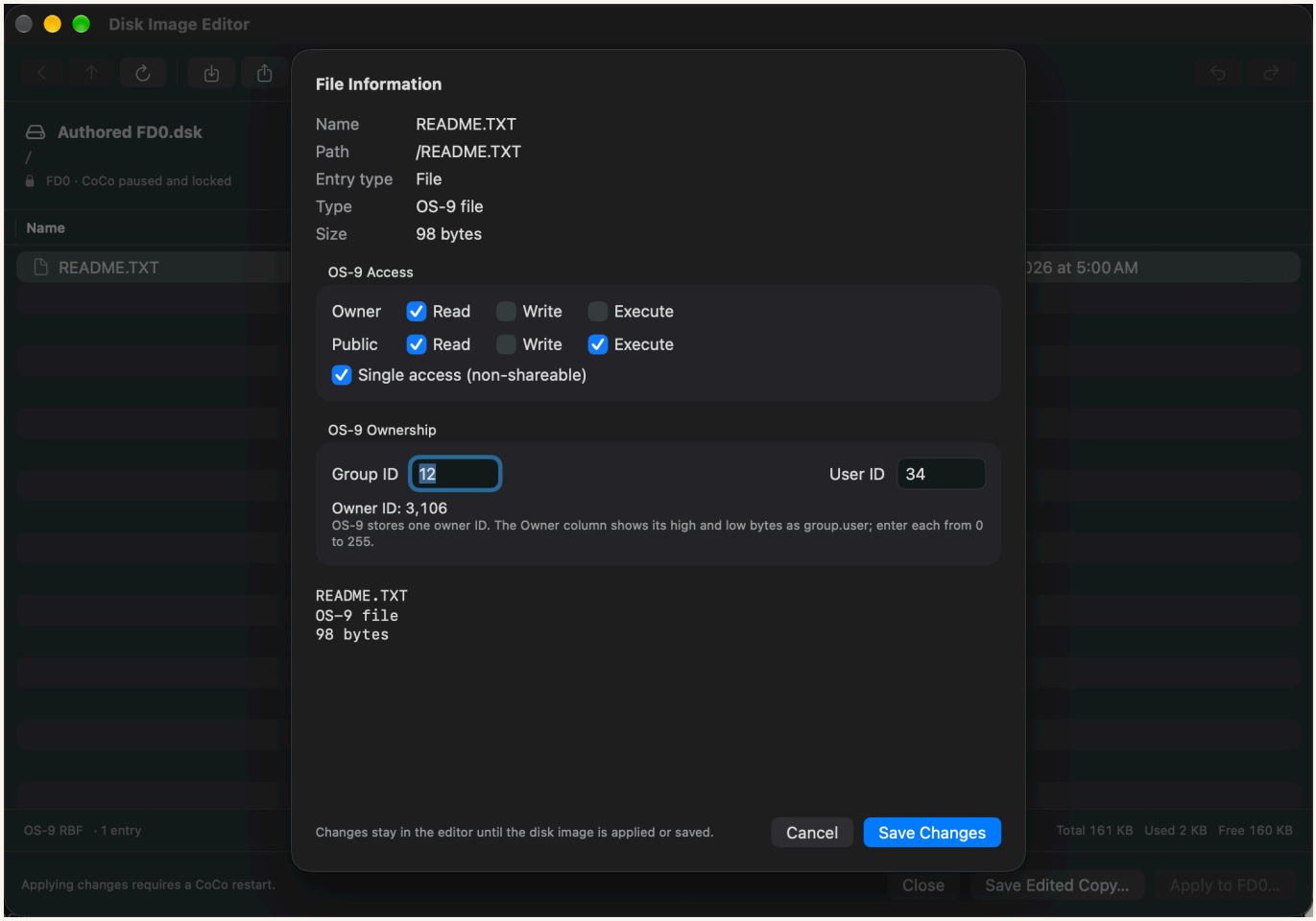


Figure 31: One OS-9 File Information sheet edits owner/public permissions and group/user ownership. These values are an example; saving the sheet updates only the working copy.

For an image with more than one recognized filesystem, the **Volume** picker selects **OS-9** or an **HDB-DOS disk** number. Disk numbers start at zero and keep their original positions even when an unformatted, unsupported or boot-reserved slot is omitted from the list. A numbered disk that overlaps declared OS-9 boot data is protected rather than offered for editing. Changing the selection visits that volume's root; it neither inserts a different disk nor starts the guest. Your private edits in the other volumes remain in the working copy. Total, Used and Free describe only the selected filesystem. **Save Edited Copy...** saves the complete image, including every other volume and preserved region.

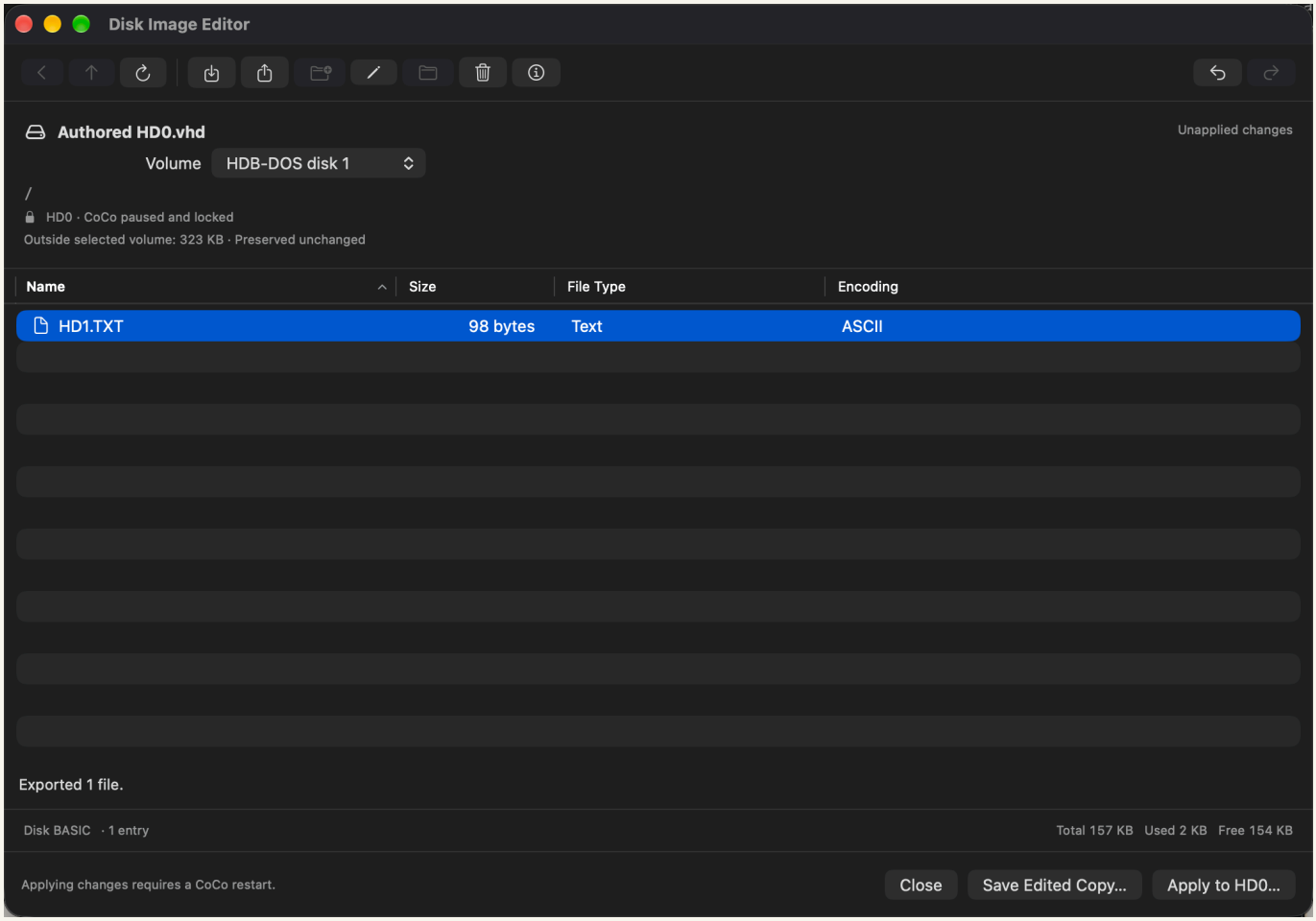


Figure 32: An authored small test image in HD0 contains multiple volumes. HDB-DOS disk 1 is selected inside HD0; its number does not mean the HD1 drive. The CoCo is paused and locked, and applying changes requires a restart. Save Edited Copy keeps the complete image.

The editor supports raw images up to **512 MiB**. OS-9 volumes use 256-byte sectors and allocation clusters of 1, 2, 4, 8, 16, 32, 64, 128 or 256 sectors. Numbered HDB-DOS volumes contain standard 35-track Disk BASIC filesystems; they begin at the start of the image or immediately after its declared OS-9 region. Arbitrary partition offsets and encoded containers such as DMK and VDK are not supported for editing. The editor does not format, repartition, convert or repair images. Individual imported or exported files are bounded to **16 MiB**.

A larger ordinary Disk BASIC floppy may contain supplementary tracks with application data. The editor preserves those bytes unchanged and excludes them from free file space. You can still add files within the normal filesystem's available space. Disk BASIC import metadata does not convert bytes: choose ASCII or Binary / Tokenized to match the file you already have.

Move files between OS-9 folders

Select one or more files and choose **Move To...** in the toolbar or context menu. Browse to a destination folder within this same disk image. Select a folder and choose **Open Folder**, or double-click it; the displayed destination path changes as you browse. **Root** returns to the disk's root, and **Up** visits the parent. Choose **Move Here** when the displayed path is the destination you want. **Command-Down Arrow** opens the selected folder, **Command-Up Arrow** goes up, **Return** activates Move Here and **Escape** cancels.

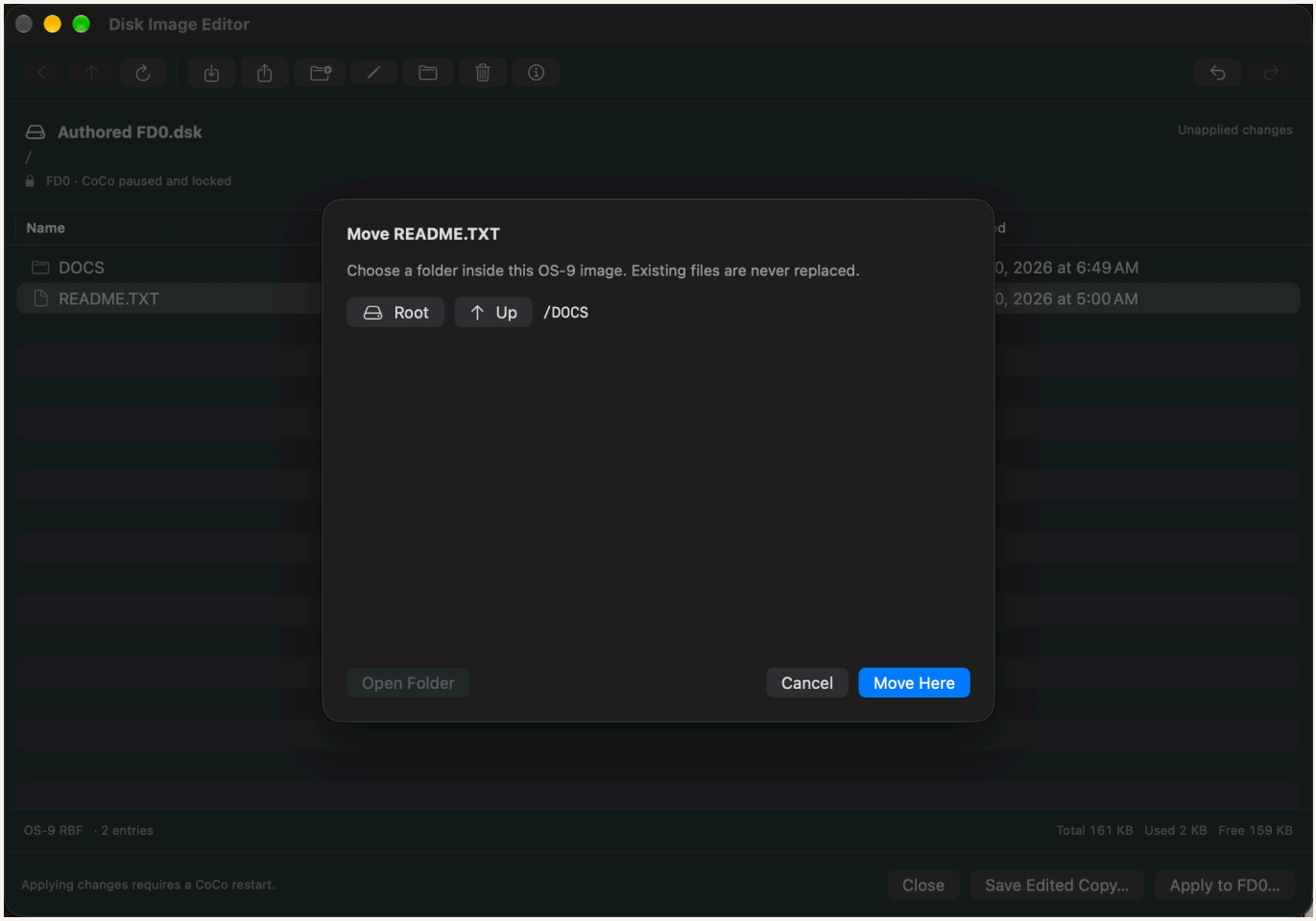


Figure 33: Move To browses folders inside the current OS-9 disk. The displayed /DOCS path is the destination; Move Here completes the move.

The chooser gives keyboard and VoiceOver users the same move operation as dragging selected files onto a folder in the table. To move a file back toward the root, use the chooser and visit the appropriate parent folder.

A move keeps the file's contents, permissions and ownership. It changes the working copy in one operation; **Undo** returns the selected files to their previous locations, and **Redo** repeats the move. As with other edits, the original image changes only when you save or apply it.

Choose a different folder from the current parent. A destination that already contains a matching filename is refused without overwriting either file; rename one of them before retrying. Folder destinations must belong to the same image. This operation moves regular files, not folders or Disk BASIC entries. A selected folder can be opened or inspected without being moved.

Send a floppy copy to a real CoCo

DriveWire for VCC Plus is a separate application for serving disk images to a real Color Computer. Install the matching public version of both apps. They keep separate windows and sessions; using one does not replace the other.

DriveWire opens the last-used personal profile and remembers its connection and disk references automatically. Its editable **Default** profile is a place to begin; **Profiles...** lets you create and explicitly switch named setups. For its complete connection, profile and recovery workflow, read the separate DriveWire companion guide.

1. Open DriveWire for VCC Plus and connect the serial session used by your real CoCo. VCC Plus does not start or reconnect that session for you.

2. In VCC Plus, open **Media → Floppy Disks...**. On the occupied source row, choose **Actions → Send a Copy to DriveWire...**.
3. Choose the receiving DriveWire session and its target **DW0–DW3** slot. Check the displayed filename if that slot is already occupied.
4. Choose **Send Copy**. On the first transfer, DriveWire asks **Allow disk copies from VCC Plus?** Choose **Allow for This Connection** or **Decline**.
5. Wait for the received confirmation. Return the real CoCo to BASIC or reset it before using changed media so its software cannot reuse old disk information.

The copy includes the source drive's current in-memory bytes, including changes that have not yet reached its Mac file. Sending does not pause or restart VCC Plus, change the source's save mode, or clear an unsaved-changes indication. The two mounted disks are independent afterward. Transfers accept compatible raw, sector-aligned disk images up to 16 MiB; an encoded track image needs a suitable raw copy. Ordinary DriveWire hosting can still use larger images.

Permission lasts only for that receiving serial connection. Disconnecting it or closing its session window removes the grant. A later send still identifies its target slot and filename; refresh if the destination changed while you decided. A disk in the shared editor or in the middle of a transaction cannot be replaced.

Before replacing a DriveWire disk, the receiver keeps its current bytes as a recovery copy. In DriveWire choose **File → Show Received Disks...**. The received copy has its own folder; **Recovery → Previous Disk.dsk** inside that folder holds the displaced image. To go back, stop guest disk activity and deliberately insert that recovery image. The original Mac source remains separate.

DriveWire's **Edit...** opens the same Disk Image Editor interface described above, with its own served-drive safeguards. Its **Software Library...** uses the same library interface restricted to disk images. VCC Plus maintains the single catalog, verified downloads and installation records; DriveWire inserts a separate writable copy. Loading instructions and hardware requirements still apply to the real CoCo. Matching public versions are released together, but each application has its own build number.

Use a virtual hard disk

In **Media → Virtual Hard Disks...**, check whether the hard-disk controller is connected, then pick **HD0** or **HD1**. It accepts raw VHD, OS9 and IMG sector images. A virtual hard disk controller has no boot ROM of its own. The **Hard Drive BASIC** and **NitrOS-9** profiles supply working combinations. Look up the guest's drive names in Which drive zero?.

The hard-disk rows offer protection, temporary changes, **Save As...**, save retry and discard for the same reasons as the floppy rows. **Edit...** opens the mounted image with the same pause, lock and restart-before-applying safeguards as a floppy. Use the editor's volume selection to distinguish an OS-9 filesystem from a numbered HDB-DOS Disk BASIC volume in an image that contains both; the guest's drive names are a separate naming system. See Edit the files inside an image. **Actions → New Disk...** asks for a capacity in KiB. It creates an empty drive in memory. Use the guest's formatting or partitioning tools to prepare it for the particular operating system, then use **Save As...** to keep its Mac file. If you want a ready-to-use disk, start from the appropriate system profile.

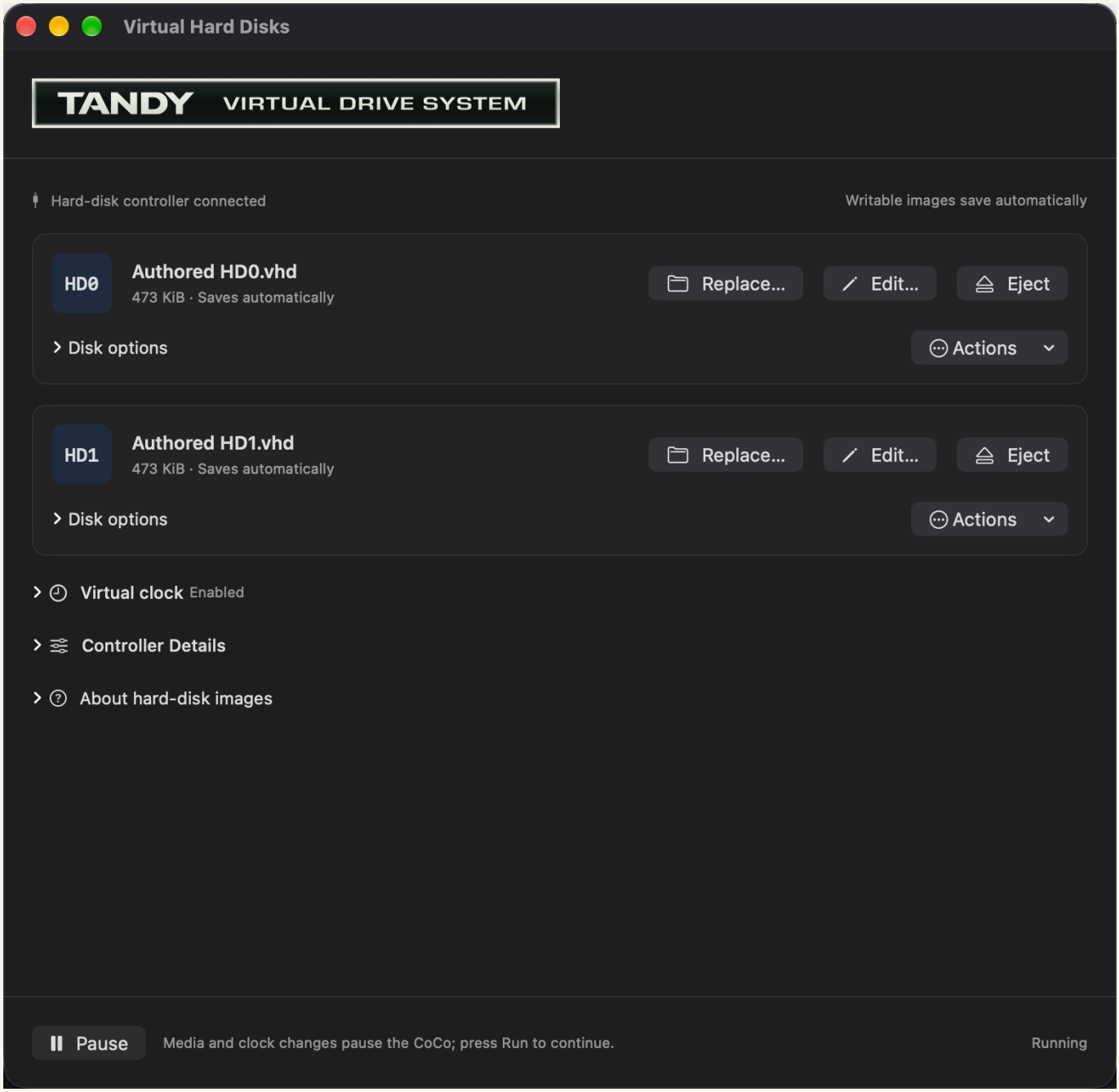


Figure 34: Virtual Hard Disks offers Edit beside both HD0 and HD1. These authored small test images demonstrate the controls; your working images may be much larger. Edit pauses and locks the machine before opening the mounted disk.

The **Virtual clock** starts from Mac local time. **Clock enabled** exposes it to supporting guest software. **Allow guest to set the virtual clock** lets the guest adjust the time for that emulated session. The Mac keeps its own clock. After media or clock changes, check for **Paused** before entering a guest command.

Follow the cartridges and card folder

A cartridge is hardware attached to the CoCo. The **Multi-Pak** is the box that gives it four slots. Open **Cartridge → Cartridges...** to choose **Single cartridge** or **Multi-Pak**, then choose a device for each slot. The app keeps the single-cartridge assignment while the Multi-Pak is in use. Floppy and virtual hard disk are separate controllers and each uses a slot; four floppy drives still need just one floppy controller. Assigning an existing controller or sound device to another slot moves it there.

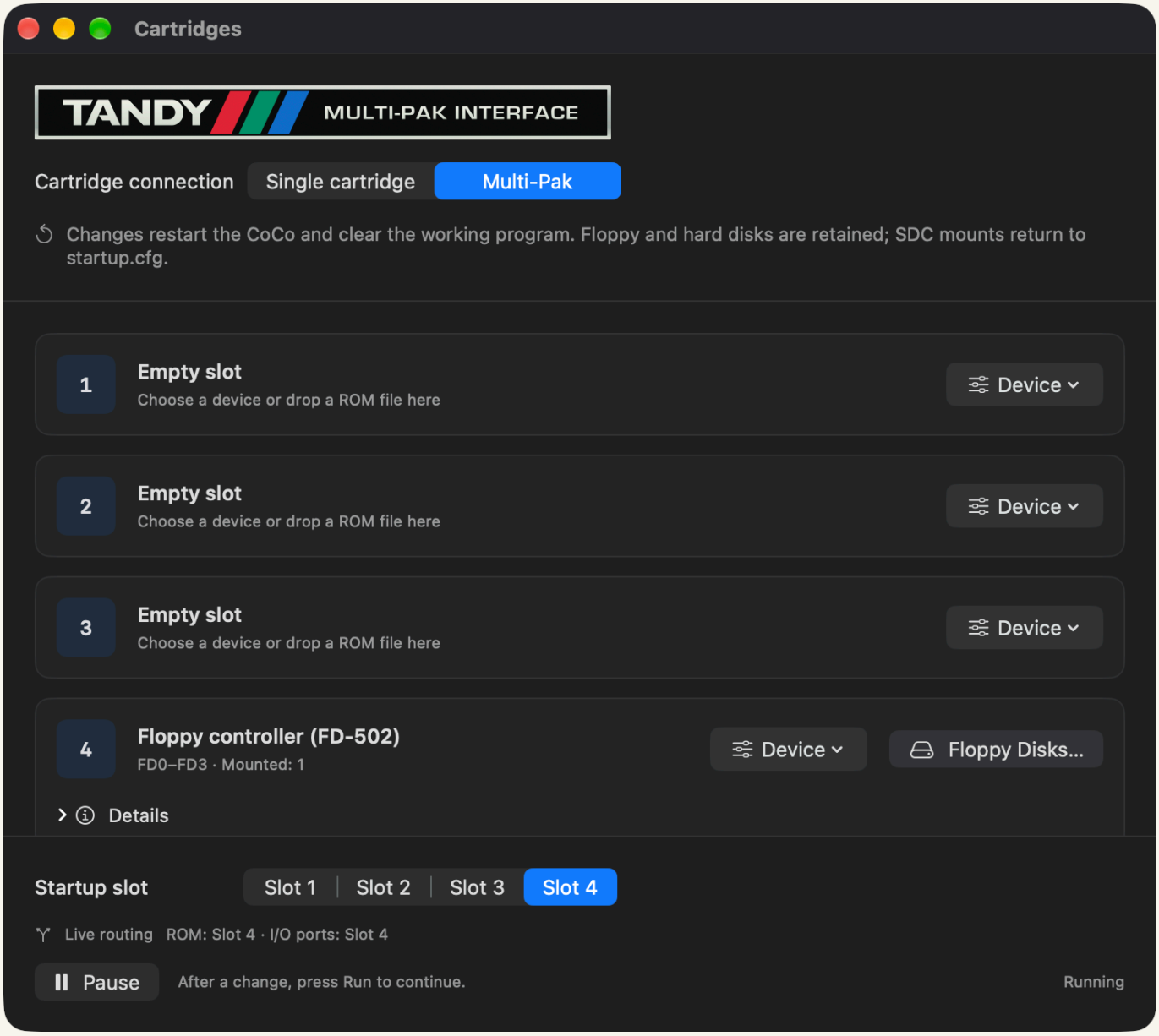


Figure 35: Cartridges and Multi-Pak shows the slot assignments and separates the startup selection from the live ROM and I/O slots.

- **ROM Cartridge...** takes the cartridge's raw program ROM, such as a .ccc game. Choose its slot as the **Startup slot** to boot the cartridge. Software can subsequently select one slot to supply ROM code and a different slot for input/output. Read the live **ROM** and **I/O ports** labels when a device appears to be in the right place but a program cannot see it. Watch the Multi-Pak for a worked example.
- **CoCo SDC** has an SDC-DOS ROM and an **SD card folder**. Use **Choose Folder...** for the directory holding your disk images and **Show SD Card Folder in Finder** to see its files. Select this slot to boot SDC-DOS. **Protect SD card files (read-only)** blocks guest writes and file-management changes; leave it on for a collection you want to preserve. With protection off, the guest can change real files in that folder. The profile remembers folder, ROM and protection. On restart, the card's startup.cfg supplies the startup mounts. SDC0 and SDC1 are on the card, independent of FD0–FD3.
- **Orchestra-90** provides stereo output to supporting software. Its optional 8 KiB ROM adds the cartridge's own music program; select that slot at startup to use it. **Speech/Sound** requires its two internal ROMs and a program that issues speech or sound commands. It has no boot program; leave the disk controller selected at startup when using it with a disk program. Expand the assigned slot to choose or replace its ROMs, then check the app's Sound and volume settings.

- The **512 KB RAM disk** works with software written to use it. Its contents survive a reset but disappear when you remove its last assignment, load a profile or quit. Save anything valuable to persistent media first. A saved profile remembers the device. Custom layouts such as this use the conventional controls.

Changing the cartridge layout cold-restarts the CoCo and clears the working program. Floppy and hard-disk media are retained, and SDC mounts return to their startup configuration. Check the slot details for missing files before loading a different profile to fix a connection.

Play or record a tape

Choose **Media** → **Cassette…** and **Insert…** to select CAS or WAV. **Play** readies the tape; the CoCo starts its motor when the guest reads it. Use **Rewind** to return to the beginning. Tape position counts bytes on CAS and samples on WAV. **Fast CAS** speeds supported ROM loads and can be changed while stopped. For the guest loading sequence, see Cue up the cassette.

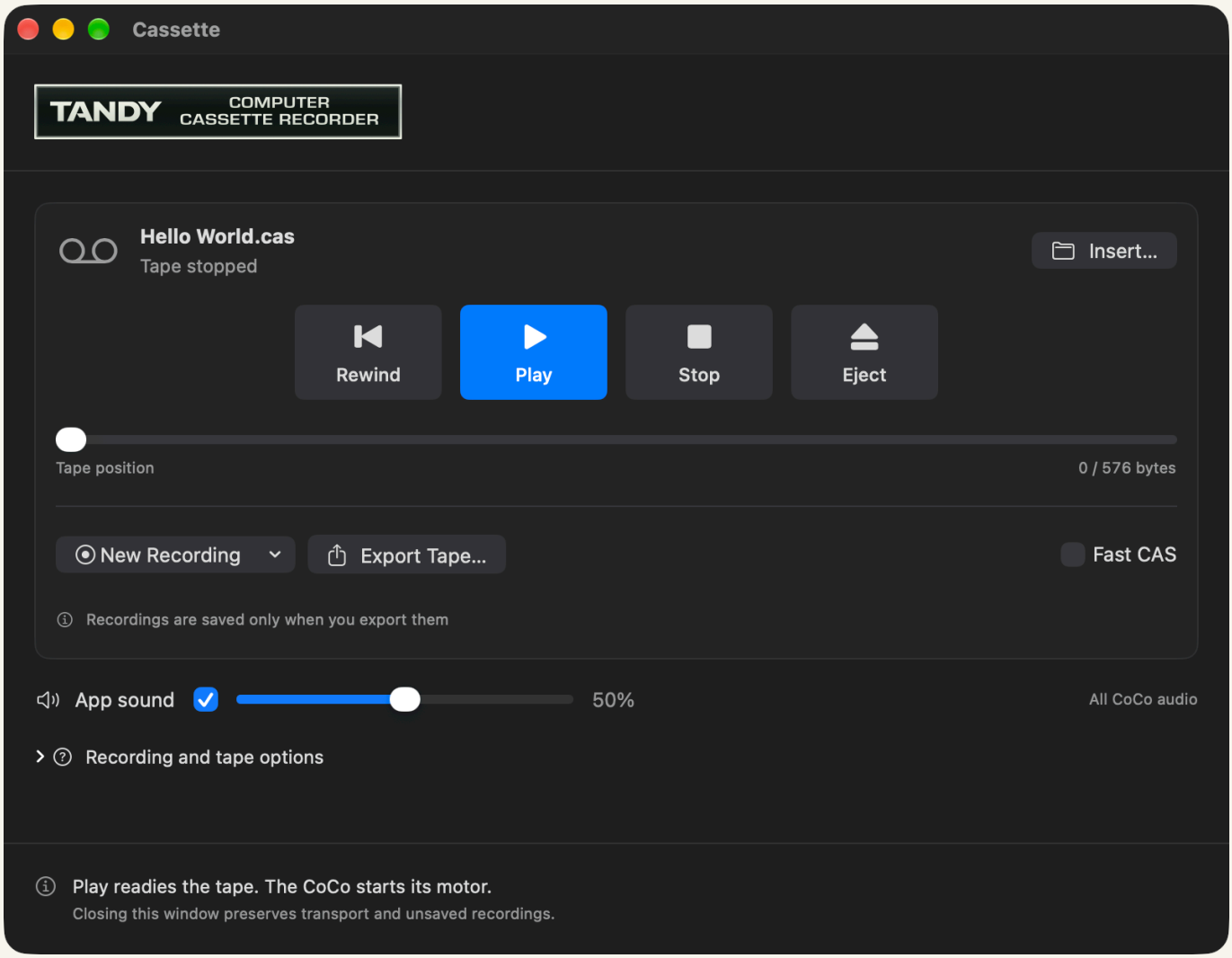


Figure 36: The cassette window shows the inserted tape, transport buttons and tape position.

Scroll below the transport to **App sound** for the Sound switch and Volume slider. They affect all CoCo audio, including cartridges, rather than just the tape. A short window can keep this row below the visible area.

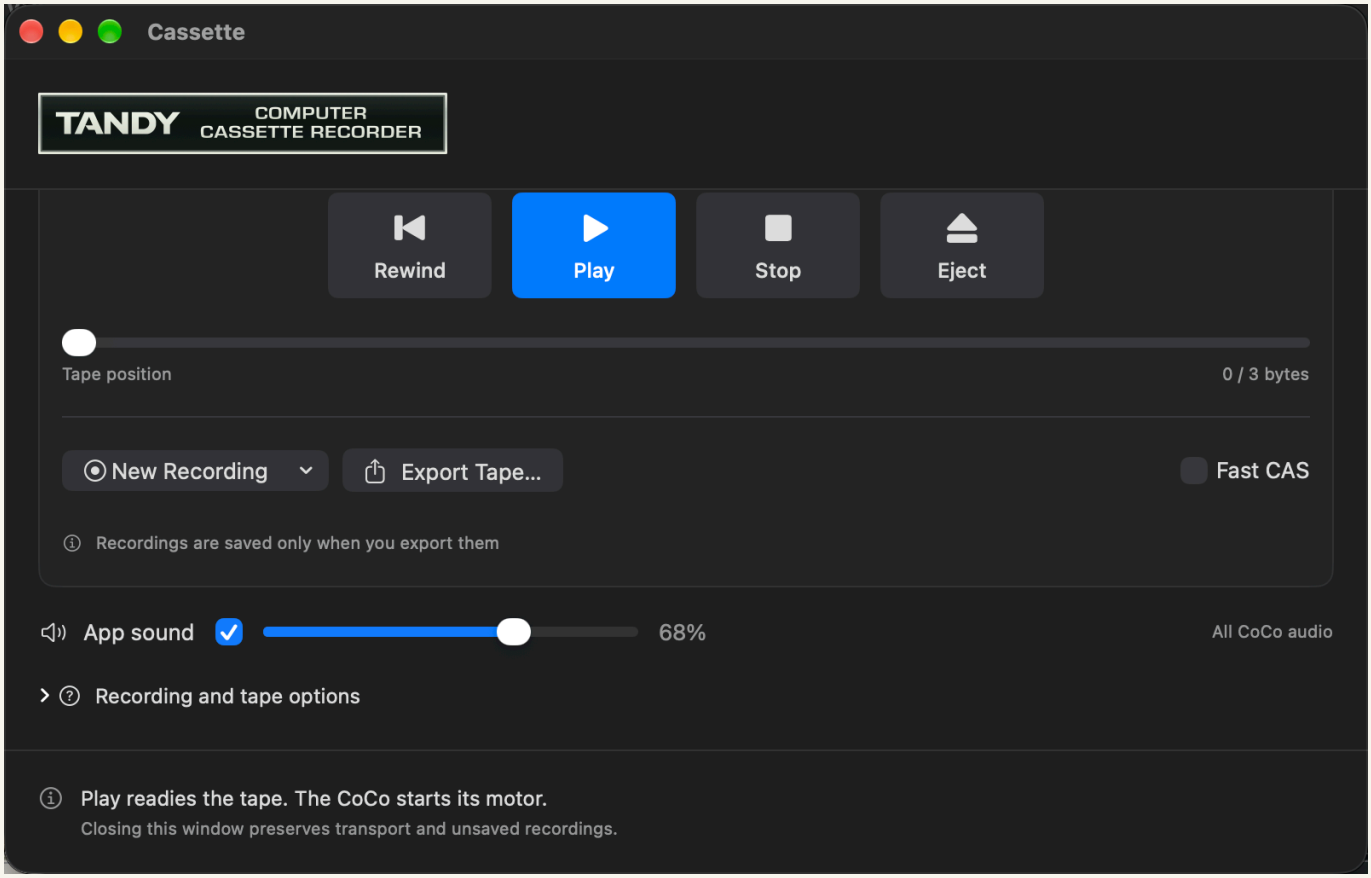


Figure 37: Below the cassette transport, App sound controls the emulator's sound and volume.

New Recording → **CAS** or **WAV** prepares a new in-memory tape in place of the inserted tape. Once the guest finishes recording, stop the transport and choose **Export Tape...** to give the recording a Mac file. **Discard...** deliberately gives up an unsaved recording. You can close the cassette window while a tape runs. Reopen it from **Media** or the tape status to check transport, position or a recording needing export.

Catch output from the printer

Click the illustrated printer or choose **Media** → **Printer Output...**. Start capture *before* asking a guest program to print. The TP-10 picture marks the CoCo's serial printer connection. The window collects the bytes the program sends there. **Follow Output** keeps the newest text in view; **7-bit BASIC Text** ignores the high bit when displaying or saving text. Control codes appear as visible markers.

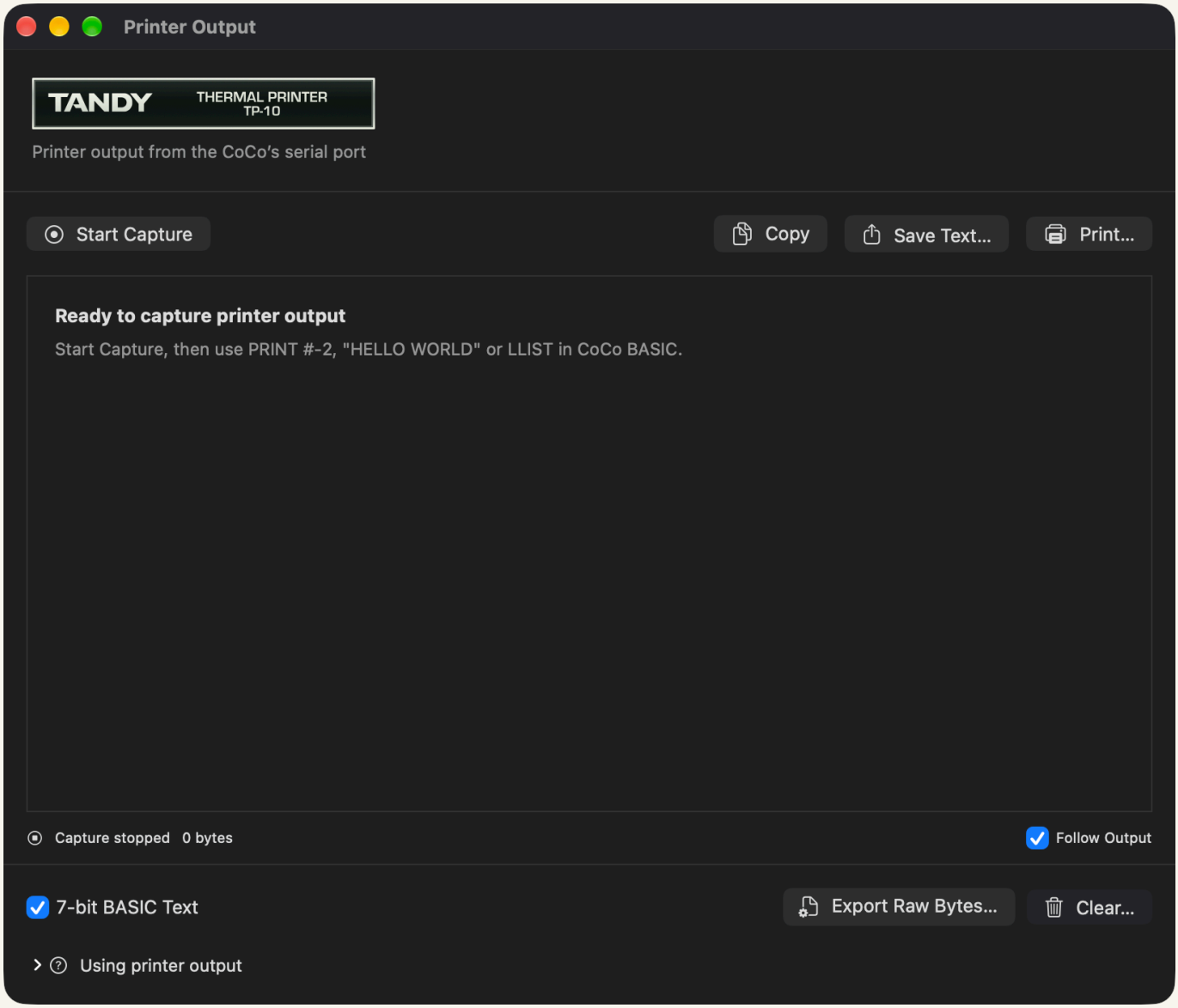


Figure 38: Printer Output offers capture, text saving and the Mac print dialog. Capture first, then ask your CoCo program to print.

Copy, **Save Text...** and **Print...** operate on all captured text; Print opens the Mac print dialog. **Export Raw Bytes...** preserves the actual bytes, including control codes. Use it for data that is unsuitable as text. The preview shows the latest 64 KiB and capture holds up to 1 MiB. When the capture fills, additional bytes are dropped: stop, save and clear before starting another. Closing the window keeps capture and its output. **Clear...** stops capture and discards the session's collected output after confirmation. A guest program that needs printer graphics may emit control codes instead of a printed picture here.

Try a print job you can count. At a Disk BASIC prompt, open an empty Printer Output window and choose **Start Capture**. Give the CoCo keyboard focus. Enter the line below, then press Return.

```
PRINT #-2,"FLIGHT TEST"
```

Open Printer Output again. The captured text reads **FLIGHT TEST** and the status shows **12 bytes**. The two words and their space account for ten characters; the print stream also ends the line. Choose **Clear...**, read the warning, and choose **Cancel**. The 12 bytes are still there. Choose **Stop Capture**. Use **Save Text...** if you want a Mac copy of the result.

Find a workbench or a reference

Chapter 9 of 10

There are three ways to write the Earth or Mars program. In BASIC you can read the question in a numbered line. In C you can find the function that asks it. In assembly you can see the machine instruction that handles a key. All three editors open from **Develop**. Each sends work to the same CoCo you see in the emulator window. Save a guest program you want to keep before loading another.

Choose an editor

- **BASIC Development…** opens numbered BASIC source. Use it to try a short program, bring a listing back from the guest, or stop before a particular statement.
- **C Development…** opens a project folder. Choose this when your work includes C source, headers, libraries or data files that must be built together. A Makefile gives the build tools their instructions.
- **Assembly Development…** opens assembly source or a project folder. Choose this when you want to work directly with processor instructions, registers and addresses. A single source file is enough to get started.

Build in C or **Assemble** in assembly creates machine code; **Run** starts the program in the guest. In BASIC the CoCo already contains the language interpreter, so the editor can load and run numbered lines directly. In all three workspaces, editing a file on the Mac and interacting with the running program use different windows. Choose **Input → Focus CoCo Display** to answer a guest prompt.

Find a word in the file you are editing

In BASIC, C or Assembly Development, press **⌘F** or click the editor's **Find in File** magnifying glass. Type the text you want. **⌘G** selects the next occurrence; **⇧⌘G** goes back. Use **Match Case** when capitals matter. To search for something you can already see, select it in the source and press **⌘E**.

Suppose you want to change a greeting in a personal copy of Earth or Mars. Press **⌥⌘F**, find the old wording and enter your replacement. **Replace** changes the selected occurrence. **Replace All in File** changes every match in this file, including matches in comments and quoted text. Check the match count first. One **Undo** reverses the whole Replace All. **Escape** returns you to the source.

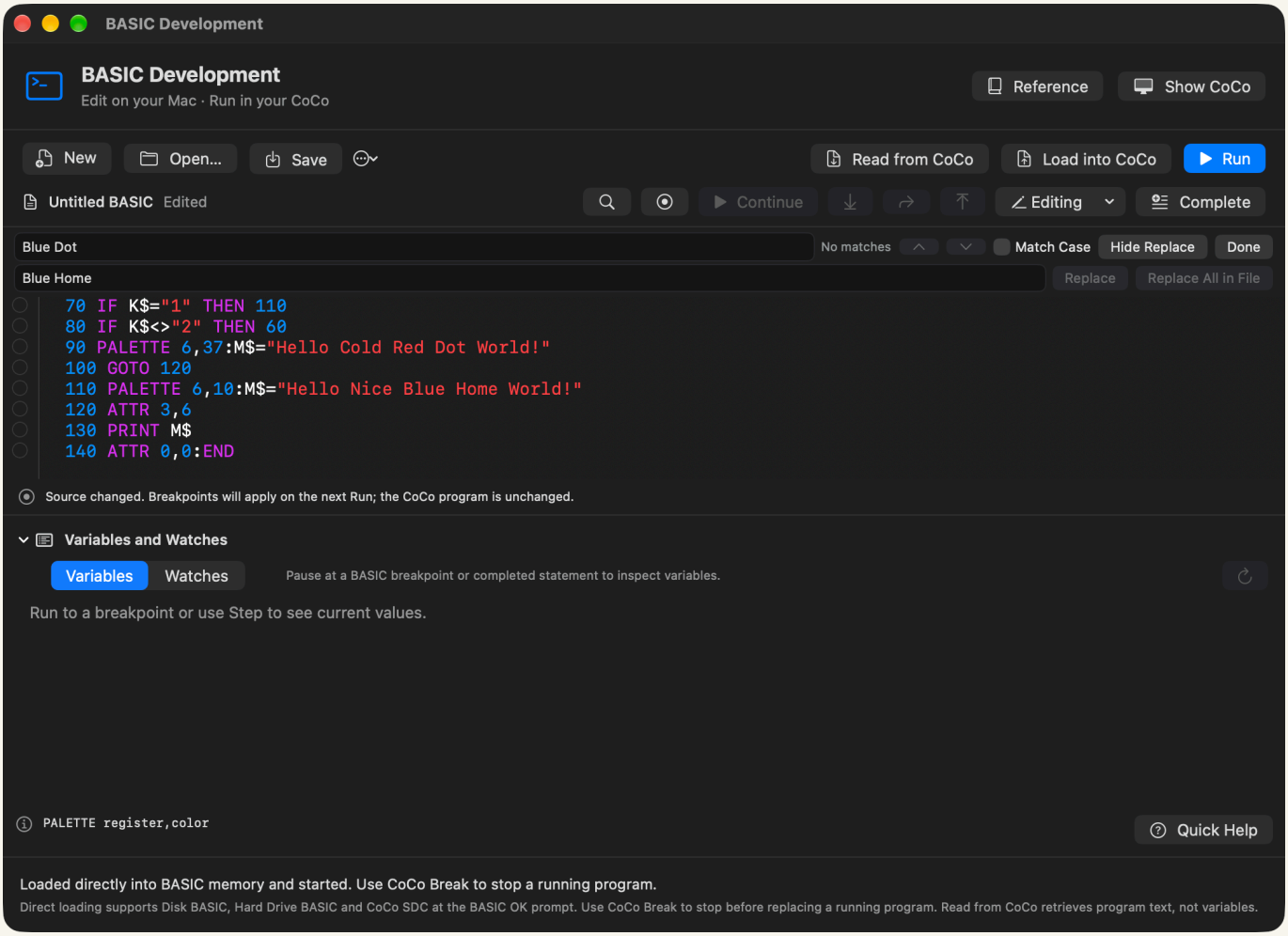


Figure 39: After Replace All, the BASIC editor shows "No matches" for Blue Dot. Line 110 now says Blue Home. The message beneath the source confirms that the running CoCo program is unchanged.

Finding works in an unsaved source buffer and in a read-only C header. Replacement requires an editable file. The change belongs to the Mac source. To see a new greeting on the CoCo, run the edited program. In C, **Find in Project** is a separate search across files. Press ⇧⌘F or use the magnifying-glass tab immediately to the right of the folder at the top of the left sidebar.

Bring a BASIC program into the editor

If you have been typing a program at the green screen, return to the BASIC prompt and choose **Read from CoCo**. You now have its listing in the editor. Click **Save** to keep a Mac source file. The listing contains the program, while variables belong to the current run. In the other direction, **Load into CoCo** transfers the editor listing and leaves BASIC ready; **Run** transfers it and starts it. Direct loading is supported at the BASIC prompt in Disk BASIC, Hard Drive BASIC and CoCo SDC. If a guest program is still running, use CoCo Break before replacing it.

Use **New** for a starter or **Open...** for a saved source file. Tab accepts a faint suggested keyword ending. Under **Editing**, **Go to Line...** finds a numbered line, **Check Source** can take you to a problem, and **Renumber...** updates the numbers and recognized references. Use the **...** menu beside Save to make a copy or export a disk or cassette. The export dialog shows which command to type in the guest. Disk export can also install a DOS bootloader so you can start that disk by typing DOS. Put VOYAGE on a disk follows a program from source into portable media.

A breakpoint is a request to pause **before** a place in the program. Set one beside a numbered line, then Run. **Continue** lets BASIC run to its next stop. **Step Into** advances by a BASIC statement and follows a subroutine; **Step Over** lets the subroutine finish; **Step Out** returns to its caller. At a BASIC stop, inspect **Variables** or add a name under **Watches**. Their values belong to that point in the program. Stop before the subtraction shows how to tell whether a line has done its work yet.

Follow a C build from source to the guest

Choose **New** for a project or **Open** for an existing project directory. The simple starter asks Earth or Mars. **Hello Disk** is the example to open when you want to read data from disk. Select source files in the project navigator. Use **Go to Definition** on a name, Back to return, and **Find in Project** to search across files. Completion suggests identifiers; parameter hints show the arguments a function expects. Headers from the installed libraries can be read in the editor.

Save All writes project edits. Select **Debug** when you want source stops and named variables. Click **Build**, read the first error under **Issues** if the build fails, then choose **Run** or **Launch Disk**, according to the project. The menu beside Build also has **Rebuild**, **Clean**, **Build Disk** and **Show Build Products**. **Build History** helps recover the output from an earlier attempt. If a source file has changed outside the editor, compare or preserve your draft before using **Reload File from Disk**; **Save Selected Copy…** gives it a different destination.

Project Settings divides the decisions into five groups:

- **Build** chooses **Generate from settings** or **Custom Makefile**. Generated projects expose the binary name, addresses, stack and release optimization. For an existing Makefile, leave the build target empty to use Make's default goal, or specify one defined by the author. Check artifact locations, clean target and parallel jobs here. Review a Makefile before running it on your Mac.
- **Sources** specifies project modules, include directories, defines and build flags. An include directory is where the compiler looks for a header named by the source.
- **Libraries** selects bundled libraries or builds project-local library subprojects. A project can depend on its own copy of a library. Check its version and build settings before substituting another copy.
- **Resources** builds data the program needs from input files and generation rules. A missing data file may let the C code compile yet leave its disk unusable.
- **Disk & Run** specifies disk contents, any DOS bootloader, and the launch method. **Quickload executable** starts a binary directly; **Mount disk in FD0** supplies a disk and can type the chosen startup commands. Use the latter when the program needs files on that disk. Review the preview before replacing an existing Makefile with a generated one.

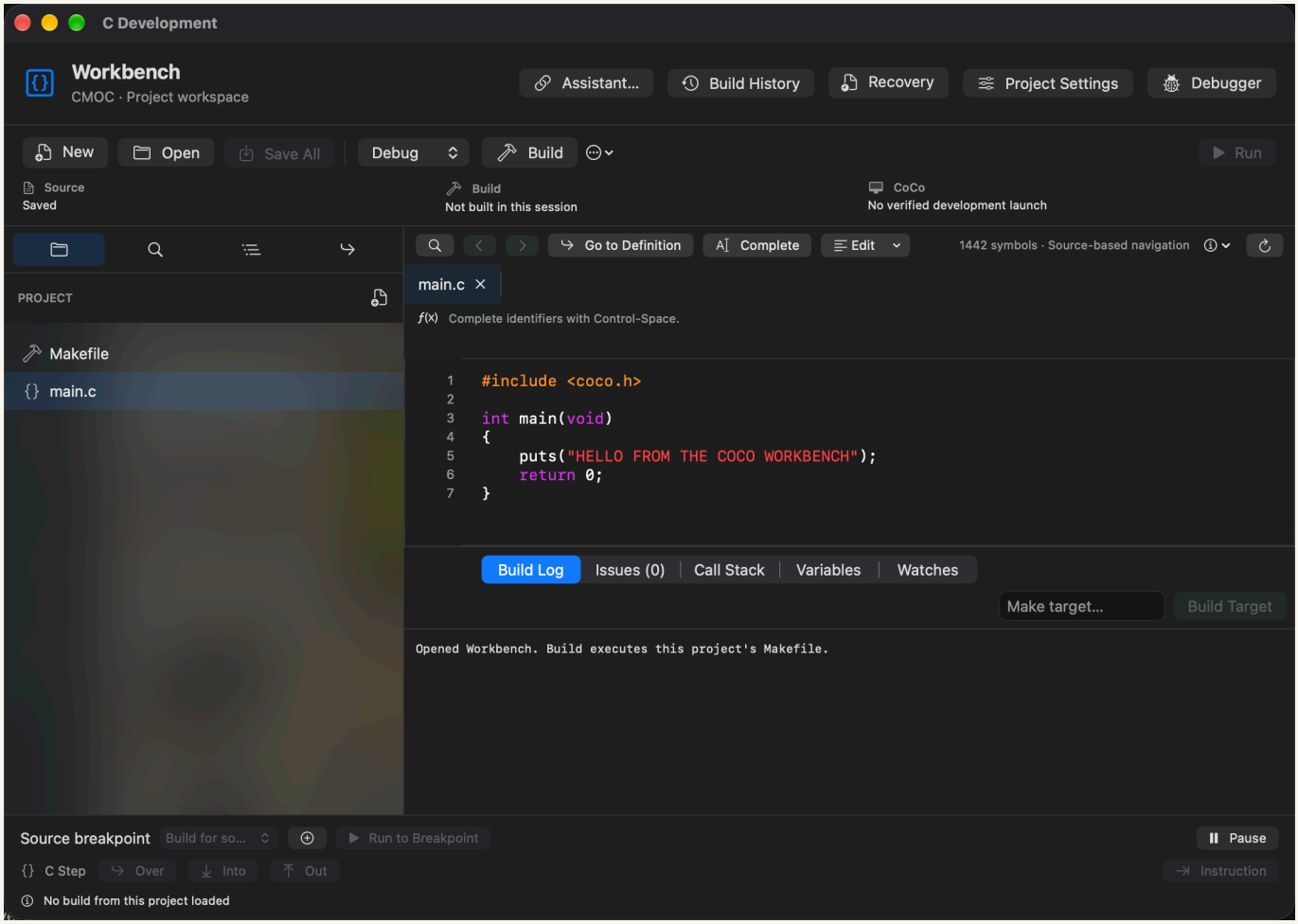


Figure 40: C Development with a small source project open. Assistant… shares the intended project; Build History, Recovery and Project Settings are separate controls. Opening the folder has not compiled or run its program.

At a C source breakpoint, step through C source or select a row in **Call Stack**. Each row represents one active function call. Select its frame to see its source, then **Variables** or **Watches** to inspect the values available in that call. Expand a structure, array or pointer to see its contents. A recursive function can have several rows, each with different locals. Execution resumes from the current stop even while you are looking at a caller. **Show Current Frame** returns from a selected caller to the active stop; the screenshot above shows an opened project, rather than a paused debug session.

For a guided first session, run the single-key C starter. For a two-file mystery, stop just after Hello Disk opens its data file.

An inherited Makefile can build a correct program without retaining enough information for C debugging. Preserve the matching object files, compiler intermediate files, listing and linker map in its Debug recipe. The debugger can then distinguish an executable function entry from a mapped C statement. An unavailable local is a limit of the retained evidence, not a value of zero. See Keep custom-Make debug evidence.

Stop at the first assembly instruction

Use **File → Open Source...** for a single file, or **Open Project Folder...** for a larger program. Select the **Target** processor. Both machines understand the 6809 instruction set; programs using 6309-only instructions need the Hitachi processor in the running machine as well. **Assemble** checks the source and creates the binary. Read **Build details** for the listing and diagnostics, or select a diagnostic to go to its location.

Choose **Debug from Entry** to stop at the first instruction, or **Run** to start without that initial stop. **Set Stop** on an instruction you want to examine, then **Continue** to reach it. **Step** lets the machine execute an instruction. An

instruction may change one register without producing anything visible on the CoCo screen. To see the moment it happens, open a three-instruction program and follow A.

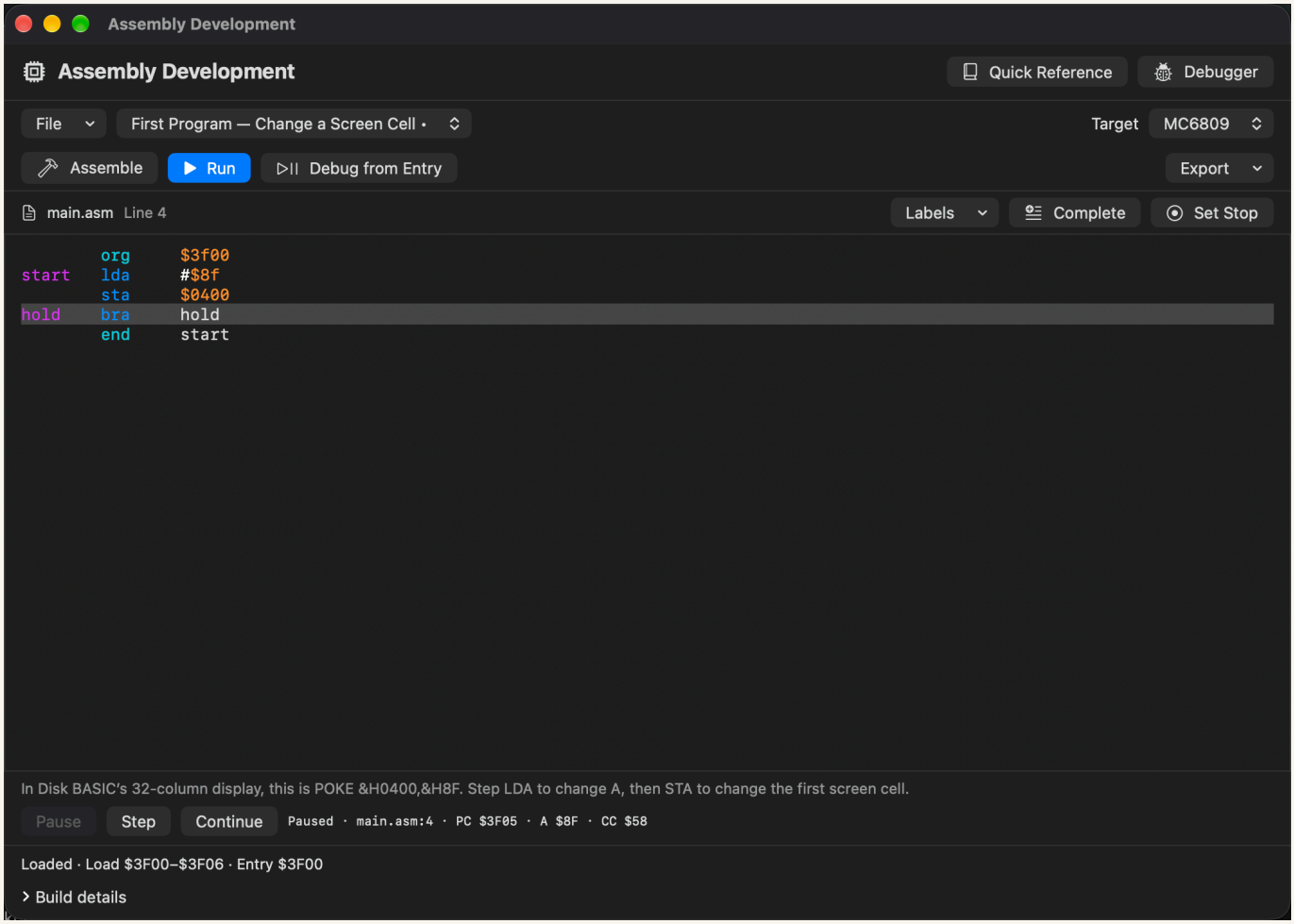


Figure 41: In Assembly Development the example has stored a character in screen memory. The highlight is at the following branch; the footer shows the current program counter and registers.

Project Settings… explains how to edit the project's settings file. It names the main source and, for linked projects, any additional modules and linker script. A custom Makefile project names its recipe, target and output binary. The app asks before executing that recipe. The native assembler-and-linker route can verify source stops against the assembled bytes; use the machine debugger to inspect the output of a custom recipe.
Export Binary… saves the binary for quickloading later. **Export Disk Image…** puts the named binary on a disk and shows the full command to load and execute it. Selecting **Install DOS bootloader** adds DOS as a second way to start it. This workflow targets a BASIC machine; an OS-9 executable uses a different format.

Look inside the machine

To try the machine debugger with a known result, catch a store one instruction before RAM changes.
Debug → Show Debugger examines the shared machine, even if the program has no source file. **Pause Execution** stops it where it is. **Step Instruction** advances one CPU instruction. A BASIC line or C statement can require many instructions. Choose the language workspace when you want to follow the listing you wrote.

- **Inspector** answers "Where is the CPU now?" Follow the program counter (PC) in disassembly, or open an assembler listing in Source view. Inspect registers or set an address breakpoint. In Memory, compare the CPU's mapped address with physical RAM. The MMU view shows how the machine maps blocks of RAM

into the CPU's address space. Navigating is safe; **Set PC** changes the next instruction, and editing a byte alters the machine.

- **Memory Tools** answers "What is in this range?" Select an address space and range to disassemble it or export the bytes. OS-9 interpretation can identify calls and module headers when examining OS-9 code.
- **Graphics Preview** answers "What would these bytes look like as pixels?" Start with the current video page. Switch to a chosen address, select an address space and graphics mode, then inspect a pixel and jump to its byte. Zoom changes the preview alone. The BASIC graphics chapter gives the two-page drawing a purpose.
- **Trace** answers "What happened just before this instruction?" Set bounds or triggers, start a new trace and reproduce the event. Stop recording, select a sample and compare its before/after registers. Markers, optional screen and scheduler events, and export help you compare two points in time.

Which debugger controls change the machine?

Go, **Follow PC**, **Show in Memory** and the MMU's RAM backing buttons change what you inspect. The MMU maps the CPU's address space in 8 KB windows. Its two task columns show programmed banks. **Effective read** tells you what currently supplies the bytes the CPU reads. A ROM overlay can hide RAM at the same CPU address. Select the RAM backing address to see that underlying memory.

A breakpoint's address is a logical CPU address in this session. Uncheck its **Enabled** box to pass it without losing its location. **Set PC Here** redirects execution when the machine resumes. **Write** in the memory inspector edits a RAM byte. Use a practice session for either change.

In **Memory Tools**, Start and End are hexadecimal and the ending byte is included. For physical RAM, the assumed CPU origin determines the targets shown for relative branches. **Disassemble** previews up to 128 rows. **Export Binary…** writes the whole selected range as raw bytes. A disk BASIC loadable binary also needs its load and execution information; for that use the assembly workspace's export.

For a graphics byte, click a point in the preview or enter its X and Y in **Pixel Inspector**. **Show Byte in Memory** opens the corresponding location. **Highlight step changes** helps locate a changed pixel after stepping.

In **Trace**, set the maximum sample count and optional start and stop addresses before choosing **Start New**. This replaces the earlier capture. Starting or stopping the trace leaves the CPU's run state alone. During review, use **Go** for a sample number and **Mark A** and **Mark B** for two points of interest. Choose **Export CSV…** for the selected sample range before clearing it or starting another capture.

Choose the right kind of help

For a BASIC command or an assembly mnemonic, choose the corresponding **Quick Reference** under **Develop** or **Help**. Search the name, check its syntax and the supported BASIC environment or processor, then open the example in its own editor document. Examples are useful even if your main project is already open.

For a worked lesson to read alongside the machine, use **Help → Manual Library…**. For a particular game's controls or an application's reference card, use the Software Library's manual or readme. **Help → Keyboard & Shortcuts…** covers Mac and CoCo keys. **About VCC Plus** and **Licenses & Notices…** identify the people and projects behind the emulator, its artwork and the tools you build with.

Work with an assistant

Develop → Connect to ChatGPT… opens the BASIC connection controls. For the same BASIC document, use **More file actions → Local Assistant Access…**. In C Development choose **Assistant…** for the open project; in Assembly Development use **File → Local Assistant Access…**. These controls share the intended document or project. Choose a read-only connection when you want advice. A review-and-run connection permits the actions named in its local consent window; inspect proposed source changes and any request to execute a custom Make recipe before approving them. The connection file is a private access credential. Keep it out of your source repository.

An existing connection keeps its original scope and advertised tools after an app update. If a newly supported feature is absent, inspect the connection's capabilities and export an appropriate new connection from the workspace. Refreshing an assistant client's tool list and approving a larger project scope are separate actions. The connection field guide explains recovery and permission choices.

For C projects edited outside VCC Plus, an assistant can request a project refresh. Clean shared tabs can update without discarding drafts. New files, changed build recipes or a wider support scope can open a native review listing the concrete changes. Approval adopts those inputs; it does not itself run Make or launch the CoCo. Resolve a conflicting unsaved draft locally before asking the assistant to rebuild.

With **Allow Disposable Playtests**, a C connection can use a separate, owned 512 KB guest, choose MC6809 or HD6309 when advertised, retain its writable test disk, and reopen that disk for save/reload checks. This leaves your shared emulator session separate. The assistant can inspect verified stops and globals, capture the actual framebuffer, and use declared graphics buffers when a project supplies the necessary build evidence. A C statement stop and a function-entry stop provide different debugging information.

Audio capture requires the corresponding permission in the exported connection. It records a bounded *following* interval of the disposable guest and advances that guest during recording; a picture capture does not advance it. The audio interval releases held keys, so a clip alone does not prove a continuously held movement test. Ask for the actual processor, build, input, images and audio timing in a test report. When a request budget is exhausted, the owned playtest can still be stopped and its disk retained. See Vibe Coding with ChatGPT for the complete workflow.

Get back to a working setup

Chapter 10 of 10

Before reaching for Restart, read the status at the bottom of the emulator. Then ask where the last good copy of your work lives. Those two observations can turn a scary moment into a small repair.

My keys do nothing

1. If the emulator window is missing, choose **File → Show Emulator** (**Command-1**). A visible BASIC editor is a different window.
2. Read the toolbar. **Run** is an invitation to resume. If a debugger deliberately stopped your program, switch back to its source window first and look at the highlighted line. Continue there when you are ready.
3. Choose **Input → Focus CoCo Display**. At a BASIC input question, type a digit and look for it *before* pressing Return. In a game, consult its manual or readme for the key or joystick port it expects.
4. If only a few keys go missing, open **Input → Keyboard and Joysticks…**. An assigned keyboard joystick can consume arrows and fire keys. If punctuation lands in surprising places, check Keyboard mode. Find the keyboard again separates window focus from input mapping.

Run and Restart are unavailable while Disk Image Editor is open. A mounted disk's editor deliberately pauses and locks the machine. Finish its private edits with the appropriate save/apply action, or close it and decide whether to discard them. Closing releases the lock and leaves the CoCo paused. Click Run when you are ready; see Edit files inside a disk image.

The picture, speed or sound seems wrong

A game is in black and white. Try **Display → Composite TV** while the game waits for input. Allow a frame to be drawn. If orange and blue appear on opposite sides from the colors you remember, try the alternate composite phase. For a CoCo 3 program designed for an RGB monitor, select **RGB Monitor**. Compare the two landing pads.

The monitor has empty space around it. Check **Display Settings…**. Integer Pixels leaves room when another whole multiple of the image would be too large. A fit-to-window choice uses more of the space while respecting the picture's shape. In System view, Expand Monitor gives some of the desk space back to the screen.

Time races or a tone goes quiet. Look for **Machine → Fast-forward**. Return to normal speed, check Run/Pause, and open **Media → Cassette…** to inspect app sound and volume. The cassette transport and app sound control different things. Your Mac's selected sound output and its own mute or volume still apply. If the game alone sounds wrong, check the title's required sound cartridge.

I cannot see the program or the drive I expected

What you see	First check	Where to go
A Software Library search turns up empty	Select **All Software**, clear filters and shorten the title. The Cover Collection shows a subset.	Find a title
BASIC lists somebody else's files	Read the active profile and the mounted filename. FD0, a hard-disk volume numbered zero and an SDC mount numbered zero can each answer a directory request in their own setup.	Which drive zero?
The filename is remembered but the drive is unavailable	Check that the device's controller is connected; then locate the original Mac file if	Floppy file safety

What you see	First check	Where to go
	it moved. Reconnect external storage or choose the correct replacement.	
The controls show a controller but a cartridge starts instead	Check the Multi-Pak startup switch and the live routing. A running program can select a different ROM or I/O slot.	See a live route change
A game ignores a moving stick	Check for activity in the Mac controller preview, then in the expected CoCo port. Look up the fire button or start key in the game's instructions.	Assign a port

A save needs attention

Leave the affected medium attached while you investigate. Read the message on its drive row. **Write protected**, **Temporary changes**, and a failure to write the backing file each call for a different response.

- **Protected.** The guest can read the image. To keep the original as a master, make a writable practice copy and attach that for future writes.
- **Temporary changes.** Edits are in the mounted working copy. Use the drive's save-to-another-file action and give the result a recognizable name *before* ejecting or changing profiles.
- **Failed save or changed file on the Mac.** If another program changed that image, save the emulator's version under a separate name so you can compare the copies. If a disconnected volume caused the failure, reconnect it and retry. Read a discard confirmation all the way through before choosing it.
- **Tape recording needs export.** Open **Media → Cassette…**, stop recording and export the tape to a Mac file. Closing and reopening the recorder window lets you return to the same transport. A cassette attached beside a program in RAM does not by itself save that program.

Busy lights can flash during a perfectly good save. Look for an actual message before taking recovery action. For a practice run in which the fate of each line is visible, see Keep your work.

A picture or movie did not finish saving

Open **Capture Settings…** from the camera disclosure or Display menu. For a failed still picture, **Save Captured Image As…** keeps the picture from the original click rather than taking a new one. A disconnected or read-only capture folder can be changed deliberately; the app does not silently send the file to a different destination.

For a movie, look under **Retained Recordings**. Try **Recover MP4** after restoring storage access or choosing a writable Recordings Folder. **Show Files** locates the retained working movie. Partial recovery may stop at the last completed fragment. Keep those files until you have checked the recovered MP4. Recording and finalizing a long movie need free space; a failed completion is not a reason to delete other media or restart the guest.

Record the CoCo picture and sound

I want my old machine back

If the Software Library changed your setup to play a title, use **Return to Previous Setup** there. If you made a personal profile earlier, select it under **Machine → Configuration Profiles…** and review its files before **Load…**. Both routes bring back equipment and media assignments. Loading the saved program *inside* the guest is a separate step.

For a familiar Disk BASIC setup, first save your Mac source, guest program, temporary images and recordings as needed. Read **Machine → Restore App Defaults…** before confirming. For a pristine starter disk you can experiment on, choose **Create Fresh Starter Media…** from the appropriate built-in profile and give the new setup a different name. Preserve your existing working disk for its saved files.

If you have already restarted and the numbered lines are gone, a matching source document or a copy saved inside an image can bring them back. Opening the same profile over and over cannot reconstruct unsaved RAM. Compare the restart choices before the next experiment.

Look up a control

Appendix A

If you remember a word from a button, search this book from **Book contents**. If you remember which menu you opened, use the directory below. When something has gone wrong and you have no idea what the control is called, start with Get back to a working setup.

Emulator window and menus

Look here	For this job
VCC Plus, **Help**, **Window**	Open Help → Manual Library… for the offline books and Ask the Manuals, Keyboard & Shortcuts for key maps, or a language Quick Reference. About VCC Plus and Licenses & Notices give the project credits and terms. Select an open window by name in Window.
Help → Archive Library...	Browse period publications or search their pages. Use **Copy Citation** to keep the source of a passage.
Help → Release History...	Read the app's official version calendar. The same history is available from **About VCC Plus**. The bundled books follow the app's official release version.
Help → Feedback...	Open the website's feedback form with VCC Plus preselected. Review the information before submitting. Software submissions and rights-holder removal requests use the Library contribution journey.
VCC Plus → Check for Updates..., **Updates & Privacy...**	Check for a newer Mac download, choose automatic launch checks, and change optional platform sharing.
File → Show Emulator	Bring the CoCo back after closing the window. Command-1 also works.
File → Open Media..., **Open Recent Media**	Choose a Mac file and its destination. Inspect the device before asking the guest to load anything.
Media → Floppy Disks... → Open Disk Image...	Browse or edit a supported disk's files without inserting it. A mounted floppy still uses the paused-machine safeguards.
File → Open ROM..., **Load Binary...**	Change the system firmware or load a compatible machine-language program. A cartridge ROM goes in **Cartridges...** instead.
Edit	Copy a guest text screen, paste text, merge or replace BASIC, or cancel a paste. Inside a Mac editor, use its ordinary editing commands.
Machine	Configuration Profiles, Machine Settings and restart decisions, Fast-forward and Restore App Defaults. You can open Machine Settings with Command-comma.
Display	Switch between System, expanded monitor and conventional Display, choose video signal, scaling or scanlines, or save a PNG.

Look here	For this job
Input	Direct keys to the CoCo or to Mac controls. **Keyboard and Joysticks…** chooses keyboard layout and port assignments.
Cartridge	Choose a single cartridge or Multi-Pak layout, devices and startup slot in Cartridges…. Attached disk controllers add shortcuts to their drive windows.
Media	Open the Software Library, recorder, printer, floppy or virtual hard-disk windows. For sound output and volume, use Cassette….
Develop	Open **BASIC Development…**, **C Development…** (⌥⌘C) or **Assembly Development…**. Choose a workbench. **BASIC Quick Reference…** and **Assembly Quick Reference…** provide syntax and examples. The C API directory is in the C manual. For all three editors, see Find and Replace.
Develop → **Connect to ChatGPT…**; **Local Assistant Access…** / C **Assistant…**	Share the intended BASIC document, C project or assembly project. Review the connection's scope, recipe permissions and optional disposable tests locally.
Debug	**Show Debugger** (⌥⌘D) opens the machine inspector. **Pause Execution** or **Continue Execution** (⌥⌘R) changes whether the CPU runs. **Step Instruction** (⌥⌘I) advances one instruction when paused. See the source listing controls or choose a language-level step.

Visible controls on the desk

Control	Where it leads
Profile name or **Profiles**	Saved setups. A modified setup can be saved separately before you load another one.
Run / Pause	Change whether the CPU is executing. At a deliberate source stop, use the source debugger when you want to continue stepping there.
Restart	Soft Reset or Cold Restart. Read the consequence for RAM before choosing.
Media, **Settings**	Shortcuts to device and machine windows. Open Printer Output or the tape recorder to keep a guest print job or a recording.
Devices, **Connections**, **Show Clickable Areas**	Find a target, reveal its wiring, or see where you can click. For a partly covered cartridge, use the Devices list.
Expand monitor, **Return to system**	Enlarge the live monitor and put it back while the program continues.
ⓘ, **FD**, **HD**, **Tape** in the footer	Read the current profile's guide or inspect the named attachment. If there is a warning, follow A save needs attention.
Wooden shelf	Browse software. The separate Manual Library is in Help.
Header **camera** and shared options	Take a PNG using the remembered None/5/10-second timer. Choose the shared Save To folder and optional video audio in the menu; only the camera starts the screenshot.

Control	Where it leads
Header **video / Stop Recording**	Record a CoCo MP4 with optional CoCo audio. Red state and elapsed time identify an active recording. Host shutter/start cues are excluded from the movie.

Inside settings and inspectors

If you know the control's name	Open this section
Advanced Filters, **Edition**, **Environment**, **Format**, **Setup**	Choose a version that fits
Apply and Restart, **Cold Restart**, **Soft Reset**	Pause, reset or restart? and the memory experiment
Build History, **Call Stack**, **Launch Disk**, **Project Settings** (C)	Follow a C build
Capture, **Follow Output**, **Export Raw Bytes**	Catch output from the printer
Clock enabled, **Allow guest to set the virtual clock**	Use a virtual hard disk
Controller Test, **Dead zone**, **Fire 1**, **Fire 2**, **Swap assignments**	Follow the thumbstick and fire button
CPU overclock, **Fast-forward**, **Present frames**	Speed, frames and sound
Create Fresh Starter Media, **Save as My Profile**, **Save Current As**	Make a personal setup
Debug from Entry, **Set Stop**, **Export Binary**	Stop at the first assembly instruction
Enable DriveWire networking, **Forget Mac Shell Permission**	Let OS-9 connect
Edit…, **Open Disk Image…**, **Restart and Apply**	Edit files inside a disk image
Volume in Disk Image Editor	Choose the OS-9 filesystem or numbered HDB-DOS disk inside a hard-drive image
Move To… in Disk Image Editor	Move selected OS-9 files to another folder in the same image; Undo restores their original locations.
Export Tape, **Fast CAS**, **New Recording**, **Rewind**	Play or record a tape
Graphics Preview, **MMU**, **Memory Tools**, **Set PC**, **Trace**	Look inside the machine
Hardware Defaults, **Memory**, **Restore Bundled System ROM**	Choose processor, memory and ROM
Install Only, **Download & Play**, **Stop Automatic Loading**	Install or play
Getting Started, **Restart & Play**	Start a curated game with its verified memory, video and loading setup
Open Source…, **Show Source Folder**	Make a personal source project and open it without building or running
Read Manual, **Online Manual**	Read a cached PDF or visit the project's online reading edition

If you know the control's name	Open this section
Library Actions, **Add Personal Software…**, **Personal Library**	Keep privately owned software on this Mac, with retained originals and writable working media.
Submit Software…	Offer a title for public inclusion or follow the removal/withdrawal link. Opening a form does not submit files.
Guides & References	Open a related walkthrough, map or software reference in Archive Library.
Manual **Bookmark this location**, **Bookmarks**	Keep one bookmark list with All Books / This Book filters. Automatic reading resume is separate.
Capture Settings…, **Recover MP4**	Choose the shared capture folder, screenshot timer and video audio, or inspect a retained recording.
Live ROM, **I/O ports**, **SD card folder**, **Startup slot**	Follow cartridges and the card folder
Read from CoCo, **Renumber**, **Step Out**, **Watches** (BASIC)	Bring a BASIC program into the editor
Read-only connection, **Review and Run**, **Allow Disposable Playtests**, project refresh	Choose assistant permissions and refresh the right project
Save As, **Temporary changes**, **Write protected** on a drive	Keep a floppy safe
Scanlines, **Smooth**, **Sharp**, **Integer Pixels**	Give the monitor more room
Update from Current, **Rename**, **Replace**, **Export** a profile	Manage the saved entry

Use an assembler listing in the machine debugger

Choose **Debug → Show Debugger → Inspector**, switch the code view from **Disassembly** to **Source**, and choose **Open Listing…**. Supply the final absolute LWASM listing for the program already loaded in the CoCo. Opening the listing changes what you see. The machine keeps its program, memory and run state.

- **Follow PC** selects a line when the program counter exactly matches a listed address. Turn it off to keep a line in view while you browse.
- **Find Next** searches source text and filenames. An addressed row has a breakpoint circle. Several listing lines can share an address, and the CPU will stop at that address regardless of which of those lines you selected.
- Control-click an addressed row for **Show Disassembly** or **Show in Memory**. Compare the listed instruction with the current mapped bytes when a stop looks wrong. **Set PC Here** changes where execution resumes and is available only when editing is allowed.

The standalone importer reads addresses from the listing and does not check the loaded bytes against it. A relocatable object listing contains section offsets that still need a linker map. Use the linked assembly project workflow or C source debugger for projects built in those workspaces. They associate the final build with the code that is launched.

Practice files

Open KEYS.bas to try the half-entered flight number. Open LANDPAD.bas to compare composite and RGB. In the app, each link opens a separate editable BASIC example. In a browser it downloads the source file. Running an example transfers its program into the emulated computer. Save any guest program you want to keep first.

The saved-line exercise works directly at the BASIC prompt. It needs an initialized, writable **practice** floppy in FD0. You can create a profile with fresh starter media first. The expected result is one saved line on disk and one later edit that exists only in RAM.

The hardware behind the controls

Appendix B

The Color Computer 3 put a working computer inside the keyboard case. A cable carried its picture to a TV or a monitor. Adding storage could mean adding two separate pieces of equipment: a controller that fit in the expansion slot, and a drive that plugged into the controller. That separation explains why there are both cartridge settings and drive settings in VCC Plus.

Historical reading

- Tandy Corporation. *Introducing Your Color Computer 3*. 1986. The opening hardware tour shows the computer, keyboard and display choices behind the illustrated desk.
- Tandy Corporation. *TRS-80 Multi-Pak Interface Owner's Manual*. 1983. The equipment setup and slot selector explain how an owner chose among cartridges. Compare the physical switch with the startup and live slot readings in VCC Plus.
- Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17, and functions, pp. 18–22. Useful beside the input and graphics exercises in this handbook.
- *Color Computer Disk BASIC Quick Reference Guide*. InterTAN Canada Ltd., 1987. Disk commands, pp. 3–8, and errors, p. 9. The BASIC disk commands operate on files inside the image attached on the Mac.
- Lindner, Tim, producer. *CoCo SDC*. Fourth version, February 2023. Hardware and software by Darren Atkinson. The card, virtual drives and command summary give the hardware background for the CoCo SDC profile.

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

Choose another book

If the two computers on your desk still seem like one, make the guest answer a question. If you have a real CoCo disk collection, take a protected disk for a practice visit. To write something you can play, fly Planet Explorer.

The pictures of controls and guest output in this handbook come from VCC Plus. Some window titles identify the isolated practice session. The app version, theme, profile or connected device can change what appears on your own desk. Historical software and its cover art in Software Library examples retain their original credits. See 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

Software, Storage, and Peripherals

Put one line of BASIC on a floppy. Carry the disk to an SD card. Record it on tape. Then change one copy and prove that the original still works.



Cream Tandy cassette recorder with a smoked tape door, counter and red record key

Brian K. Holman

Book 5 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Put a program somewhere 3
Take the same disk to an SD card 8
Find the floppy behind drive zero 12
Record a program you can rewind 17
Choose what goes in the slot 20
Install, play, and come home 23
Send a receipt, a tune, or a voice 27
Leave the original just as you found it 31
What have I got? 37
Beside the original hardware 40
Publication and artwork notices 41

Put a program somewhere

Chapter 1 of 8

You're about to make a disk small enough to send to a friend. It will hold a program called **CARGO**. When your friend runs it, their CoCo will say **CARGO ORIGINAL**. They need neither your running emulator nor the contents of its memory. What, exactly, do you have to send them?

Give the disk a home on the Mac

1. Make a new folder in Finder for this book's exercises. Choose a place where you can find it again. Select the empty, initialized practice disk. Save a copy in your folder as **CARGO.DSK**. In the app reader the link opens a save panel. In a browser, rename the downloaded copy in Finder. If you opened the book directly from its folder, you can also duplicate **media/BLANK.DSK** into your practice folder.
2. Start in **Disk BASIC**. If a program is already in RAM, save anything you want to keep. To change setups, choose **Machine → Configuration Profiles…**, select **Disk BASIC**, then **Load…**. Read and confirm the restart. Resolve any pending work before switching.
3. Choose **Media → Floppy Disks…**. Find **FD0**. Use **Insert…** on an empty drive, or **Replace…** on an occupied one. Choose the **CARGO.DSK** you just made. Read the filename in the FD0 row. On this disposable copy, leave **Write protected** and **Temporary changes** off under **Disk options**. The row should say **Saves automatically**.
4. Resume the emulator with the Mac toolbar's **Run** if it paused. Click the CoCo screen, or choose **Input → Focus CoCo Display**. Wait for the **OK** prompt.

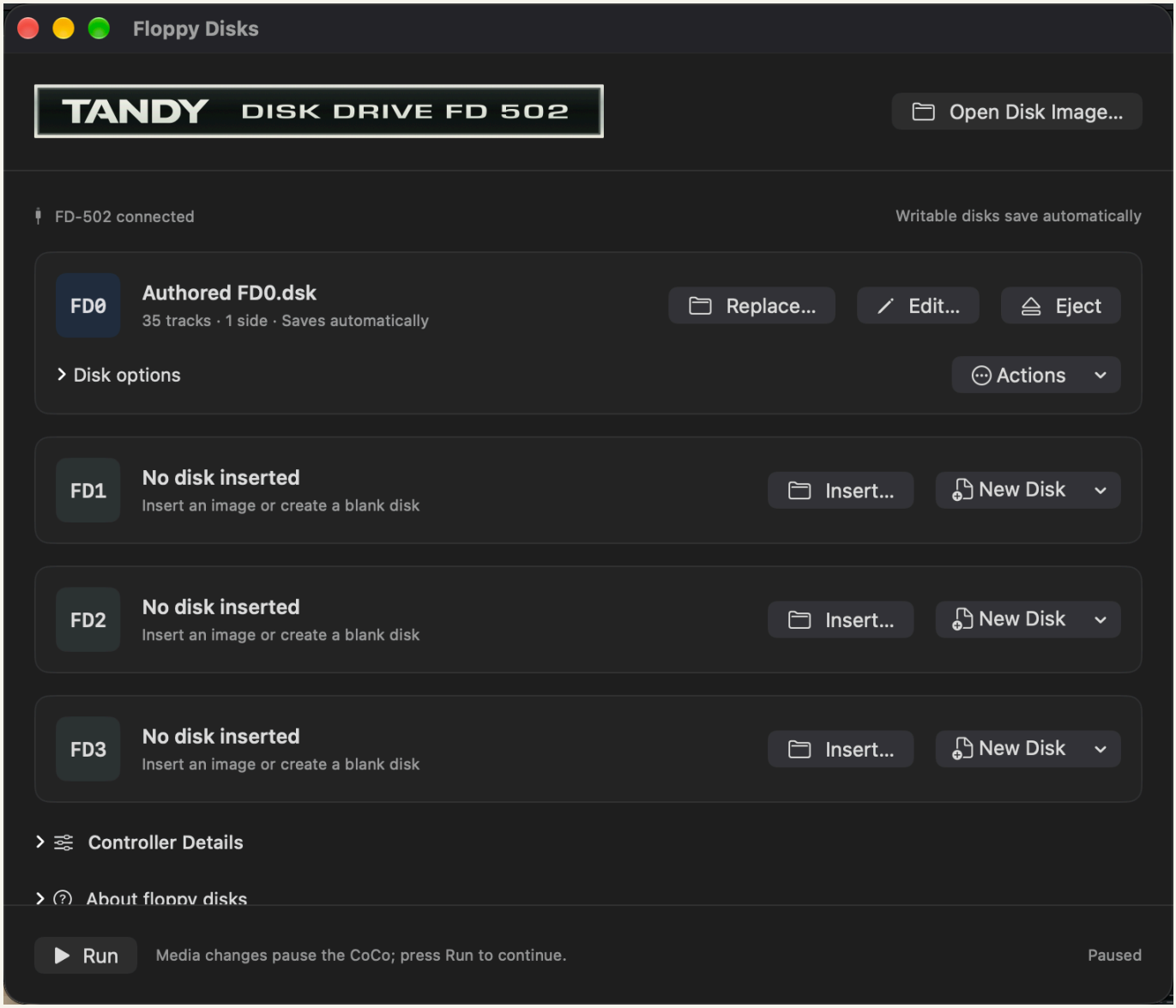


Figure 1: The Floppy Disks window puts the Mac filename and save mode beside the FD0 label. This example uses an authored test disk; look for CARGO.DSK on yours.

In the original disk system, a cable ran between a controller in the cartridge slot and the drive. The FD-502 controller understood the computer's requests to read and write. The physical drive held whichever floppy you put in it. Here, the device named **FD0** plays the part of that first drive. The image in it can change while the drive keeps its name. One FD-502 can serve FD0 through FD3.

Put a line in the directory

At the BASIC prompt, type `NEW` and press Return to begin with empty program memory. Type the following line and press Return. You can also open the source in its own editor to see it beside the machine. For this exercise, enter the line on the CoCo so you can follow the disk operation yourself.

```
10 PRINT "CARGO ORIGINAL"
```

A numbered line goes into memory. Now type these two commands, pressing Return after each one and waiting for **OK**.

```
SAVE "CARGO:0"
DIR 0
```

Find **CARGO** and **BAS** together in the list. That is a BASIC file inside the emulated disk. The zero in the SAVE name selected guest drive zero. In the Disk BASIC profile, that drive is FD0. The new Mac file you should see in Finder is the **CARGO.DSK** you chose earlier. You can store several BASIC files inside one such image.

If someone asks for…	Give them…
A line of BASIC to read or edit on the Mac	The editable **CARGO.bas** source
The floppy they can put in a drive	A copy of **CARGO.DSK**, after saving finishes
The name to type at the CoCo prompt	**CARGO**, the BASIC program in the disk's directory

Bring it back after NEW

Before clearing RAM, check FD0 for a completed save. If its row asks you to save pending changes, resolve that first. Enter the following, one command at a time:

```
NEW
LIST
LOAD "CARGO:0"
RUN
```

After **LIST**, you should see **OK** with no numbered lines. After **RUN**, you should see **CARGO ORIGINAL** and **OK**. The Mac **Run** button resumed the processor earlier. The word `RUN` here asked BASIC to execute the program it had just loaded. The result shows that the disk carried enough information to rebuild the program after RAM was cleared.

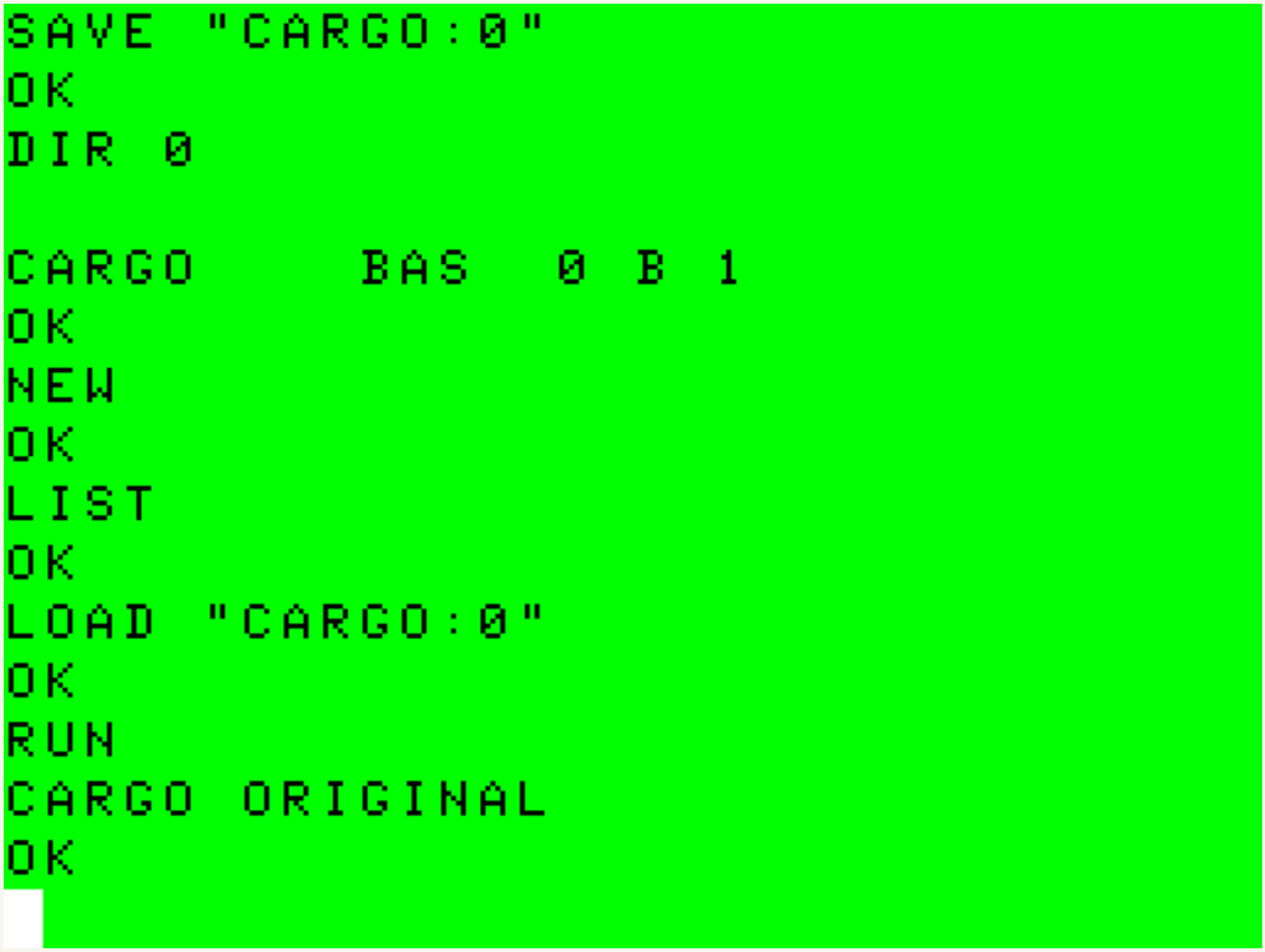


Figure 2: The guest first lists CARGO BAS on drive 0. After NEW and an empty LIST, LOAD and RUN recover CARGO ORIGINAL.

When the drive has finished, eject **CARGO.DSK** in **Floppy Disks**. The disk stays in your Finder folder. You'll copy it to a different kind of device in the next chapter. If you're picking up the book at that point, you can use the finished CARGO disk instead.

When someone hands you a different disk

Our practice file is a standard 35-track, single-sided Disk BASIC image. Old collections can also hold JVC sector images, DMK track images and VDK images. A DMK can retain details of a track that a plain list of sectors leaves out. The FD-502 sometimes recognizes DMK or VDK contents even when the name ends in .dsk. **Save As…** preserves the mounted representation. Renaming an image changes its label in Finder.

Start by mounting a copy with **Write protected** enabled. A protection flag inside a DMK or VDK header may lock that choice on. Check the software's own loading instructions before assuming Disk BASIC can list its directory. An OS-9 disk can fit in a CoCo drive and still require OS-9 to interpret its files. The same distinction applies to software written for a different computer.

An initialized disk already has a directory ready for use. **New Disk** in the Floppy Disks window also offers Standard DSK, DMK Track Image and VDK Disk Image, but those begin as unformatted media in memory. A guest format prepares their directory and destroys any previous contents on the selected disk. This book supplies an initialized blank so you can learn to save and reload without formatting a drive. Use the media field guide when deciding what to do with an unfamiliar file.

Inspect the files from the Mac

Choose **Edit...** beside the occupied FD0 row to open **Disk Image Editor**. The CoCo pauses and is locked while that mounted image is being edited. The table shows files inside the disk, with actual allocation totals below it. Exporting a tokenized BASIC file preserves its disk bytes; it does not turn it into the editable text listing used by BASIC Development.

Import, rename and delete work on a private copy and support Undo/Redo. For a Disk BASIC import, choose the file type and ASCII or Binary / Tokenized metadata to match the bytes you already have. These choices do not translate text, tokenize BASIC or assemble a binary. **Save Edited Copy...** keeps a separate image. Applying to the mounted drive requires **Restart and Apply**, which clears guest RAM before it can use the changed filesystem. Closing without applying leaves the source unchanged and the CoCo paused.

For a separate file, **Media → Floppy Disks... → Open Disk Image...** opens the editor without mounting it. A supported raw OS-9 image can be browsed too, including folders and its File Information permissions. Editing has narrower format limits than mounting: DMK, VDK, unsupported OS-9 geometry and oversized images may mount normally yet be refused by the editor. Use the handbook procedure for those limits, protection and save choices.

Carry the disk to a real CoCo

If your real CoCo uses DriveWire, open the separate **DriveWire for VCC Plus** app with its remembered personal profile and connect its serial session. Its separate companion guide explains profiles, connections and recovery. Back in VCC Plus's **Floppy Disks** window, choose **Actions → Send a Copy to DriveWire...** on the occupied row. Pick that session and a DW0–DW3 slot, then choose **Send Copy**. DriveWire asks **Allow for This Connection** before accepting the first transfer on that connection.

This sends the current disk bytes, including guest changes still in memory. It does not save or modify the emulator's source image, pause the emulator or clear its unsaved-changes indication. The destination becomes a separate working copy. If a disk was already there, DriveWire retains its current bytes for recovery before replacing it. Stop guest disk activity and return the real CoCo to BASIC or reset before using changed media. Do not assume the physical CoCo's cached directory changes just because the Mac switched images.

DriveWire also reuses VCC Plus's editor and a disk-only Software Library. There is one catalog maintained by VCC Plus; you do not update another database in the DriveWire app. Install matching public versions of the two apps. For permission, format limits and the **Show Received Disks...** recovery folder, follow Send a floppy copy to a real CoCo.

Take the same disk to an SD card

Chapter 2 of 8

A CoCo SDC cartridge lets a Color Computer open disk images kept on an SD card. In VCC Plus a folder on your Mac plays the part of that card. Put yesterday's disk image in the folder, and you can use it in a second machine without retyping a line of BASIC.

Prepare a practice card

1. Have **CARGO.DSK** ready in Finder. Use the ejected disk from Put a program somewhere, or save a copy of the finished disk. The guest will look for those short names, so keep the filename **CARGO.DSK**.
2. Save the program in RAM and finish any pending media changes. In **Machine → Configuration Profiles…**, select **CoCo SDC** under **System Profiles**. Choose **Create Fresh Starter Media…**, give the personal profile a name such as *Cargo Card*, then select and load that profile. Creating a fresh set gives you your own card folder. Loading the profile restarts the guest.
3. Choose **Cartridge → Cartridges…**, find the **CoCo SDC** slot, and open its details. Choose **Show SD Card Folder in Finder**. You can reach the card controls from the cartridge in System view, too. Copy **CARGO.DSK** into the folder alongside **HELLO.DSK** and **startup.cfg**. Keep all three as separate files.
4. For this reading lesson, enable **Protect SD card files (read-only)**. Resume the machine if paused. Focus the CoCo display and wait for the SDC-DOS **OK** prompt.

A fresh card starts with a disk named **HELLO.DSK** in its first virtual drive. Making a personal profile alone can still point to shared media. It's the **Create Fresh Starter Media…** choice that gives this experiment its own files.

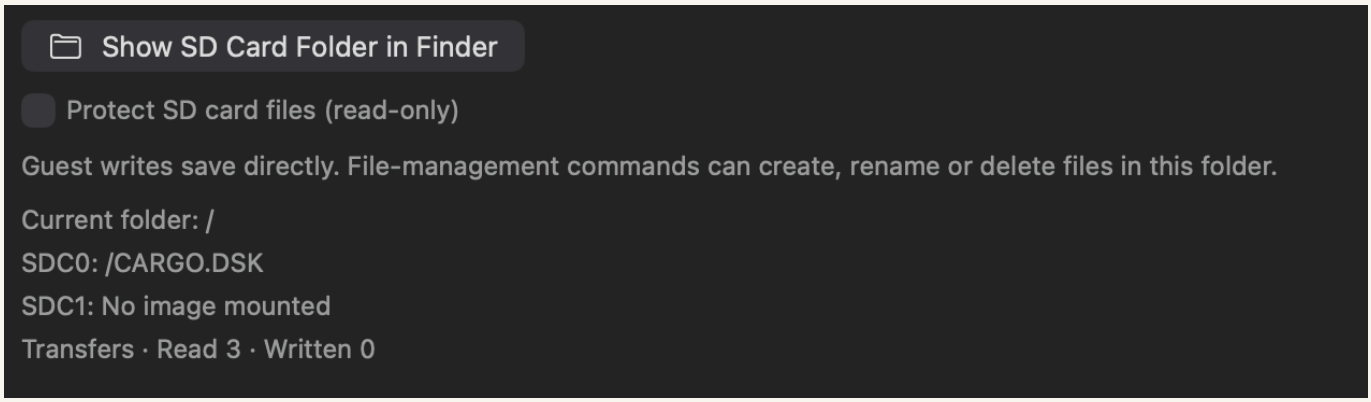


Figure 3: The native cartridge details show CARGO.DSK mounted in SDC0, an empty SDC1 and the card folder at its root. In this capture protection is off; turn it on for the reading exercise.

Read the card and then read a disk

Enter `DRIVE` at the SDC-DOS prompt. On a fresh startup the first row should name **HELLO.DSK**. Now enter `DIR –` with a space and a minus sign. Look for **CARGO DSK** and **HELLO DSK**. This is the directory of the card's current folder. You might recognize it from the Finder window. SDC-DOS displays a filename and its extension in separate columns.



Figure 4: DIR followed by a minus sign lists the disk images CARGO and HELLO and the startup file in the emulated card folder.

Mount your disk, then ask two different questions about it. Type each line separately.

```
DRIVE 0,"/CARGO.DSK"
DRIVE
DIR 0
```

DRIVE now lists **CARGO.DSK** in position 0. **DIR 0** looks *inside* that image and lists **CARGO BAS**. Try explaining those two results to someone who has only seen the Finder folder. Which name would they need to mount? Which would they need to run?

```
LOAD "CARGO:0"
RUN
```

Watch for **CARGO ORIGINAL** and **OK**, just as on the FD-502. The disk and its BASIC file kept the same contents through the move. The device interpreting drive zero changed. In this setup, use the SDC cartridge's mount display or the guest's **DRIVE** command to see the image you selected.

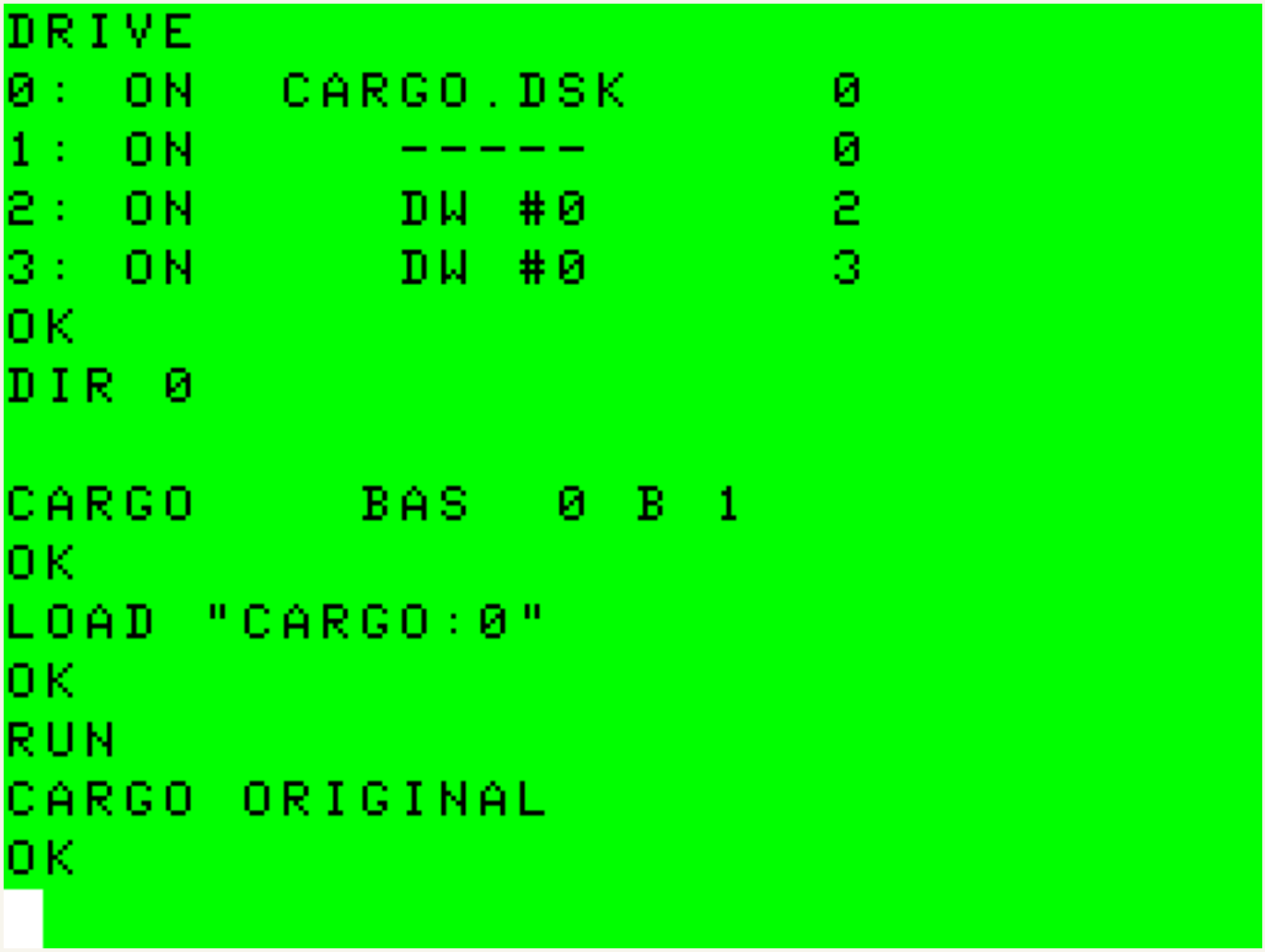


Figure 5: SDC-DOS shows CARGO.DSK on drive 0, DIR 0 shows CARGO BAS, and the loaded program prints CARGO ORIGINAL.

Keep a second image handy

The card supplies two image drives. Leave **CARGO.DSK** on SDC0 and put the starter disk on SDC1:

```
DRIVE 1,"/HELLO.DSK"
DIR 0
DIR 1
```

Look for **CARGO BAS** in the first directory and **HELLO BAS** in the second. Type `DRIVE` again to inspect the two mounted filenames. In the SDC-DOS 1.75 listing you may see additional entries marked **DW**. The two SD image mounts you used here are rows 0 and 1.

The slash at the start of each path means the top of this card. The Mac folder itself can have a long name with spaces. For files the guest must find, use short names of up to eight characters plus an extension of up to three, as in **CARGO.DSK**. If the intended image is missing from `DIR –`, start by checking **Show SD Card Folder in Finder**. Copying the image into a neighboring Mac folder can leave a perfectly good disk out of reach.

Decide what the card remembers

Open the starter card folder in Finder and examine **startup.cfg** with Quick Look. The fresh file asks the card to mount **HELLO.DSK** as drive zero when it resets. The commands in this chapter changed the current mounts for

the running session. Leave the starter file in place, and the next restart can bring HELLO back even though CARGO is still stored in the folder. Your profile remembers the folder's location. Keep the folder itself when moving a collection.

Some software expects a whole family of disk images beside one another. The CoCo SDC edition of *Double Dragon Arcade Port*, for example, comes as nine disks and uses SDC-DOS to switch among them. When you encounter a set like that, keep the filenames and directory relationships in the author's instructions. The Library's text readme for that edition explains the first mount and the game controls. There's no need to download the game for the CARGO lesson.

Leave a collection you only want to read protected. Turning off card protection lets guest software write disk images and manage actual files inside that Mac folder. Give an experiment its own copy first. This differs from FD0's Temporary changes, where you can choose when to write one mounted floppy image back to its Mac file.

Find the floppy behind drive zero

Chapter 3 of 8

A BASIC program on the hard disk and a BASIC program on a floppy can both be called **HELLO**. You could list drive zero, see a familiar filename and still be about to save on the wrong disk. Let’s give just one of the two disks a file you can recognize.

Set up two destinations

Save anything you want to keep from the current session. In **Machine → Configuration Profiles…**, select **Hard Drive BASIC** under **System Profiles**. Choose **Create Fresh Starter Media…**, give the new personal profile a name such as *Cargo Volumes*, then load it. Creating the files and loading the machine are separate actions. Using fresh media here gives you a safe place to write a marker and lets you compare your directories with the ones in this chapter.

Open **Media → Virtual Hard Disks…**. Find the assigned Mac image beside **HD0**. Now open **Media → Floppy Disks…** and find the separate image beside **FD0**. Both can be ready at once. The hardware setup includes a virtual hard-disk controller serving HD0 and HD1, and a floppy controller that supplies the disk ROM and serves FD0 through FD3. The profile selects compatible firmware for you. Resume and focus the CoCo. You should see **HARD DISK BASIC** at startup and reach **OK**.

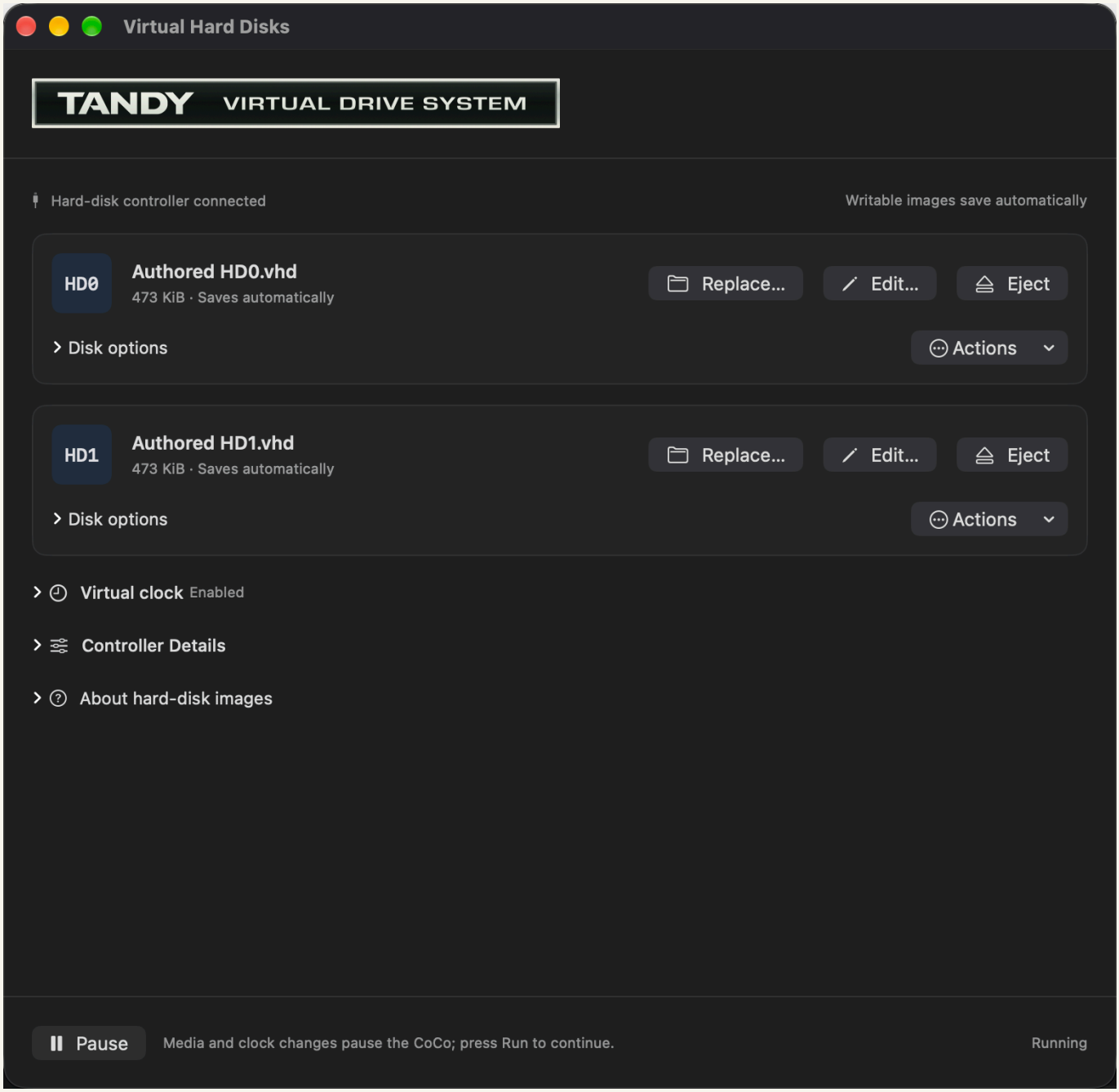


Figure 6: Virtual Hard Disks gives HD0 and HD1 their own assignments and Edit buttons. These authored small images illustrate the controls; use your personal profile's working images for this lesson. Compare HD0 with FD0 in the separate Floppy Disks window.

A floppy image represents one disk. A hard-disk image for this profile contains room for many numbered BASIC disks, called *volumes*. Here **HD0** is the Mac app's name for the first hard disk. **Volume 0** is one BASIC disk *on* that hard disk. Keep these two names apart, especially if your collection also has a second hard-disk image.

Leave a marker on volume zero

Enter the following lines, pressing Return after each. They assume you just loaded the fresh Hard Drive BASIC profile, where BASIC drive zero initially reaches volume 0 of HD0.

```
10 PRINT "VOLUME ZERO"
SAVE "VZERO:0"
DIR 0
```

Look for **VZERO** alongside **HELLO**. Write the two names down if you like. Call this your *hard-disk directory*. In the example, **FREE=66** follows the listing. Another disk may have a different amount of free space. The filename you added is the evidence to follow.

Ask the floppy the same question

At the next prompt, enter:

```
NEW
DRIVE OFF
DIR 0
```

Compare this directory with the first one. On the fresh floppy **HELLO** is there and **VZERO** is missing. **NEW** cleared the marker program in RAM. **DRIVE OFF** is the RGB-DOS instruction that lets ordinary floppy drives answer at numbers 0–3. FD0 just answered the same `DIR 0` you typed before. Both drives are still visible on the Mac. You can watch their read indicators as you issue the commands, but the marker stays in the directory long after a light goes out.

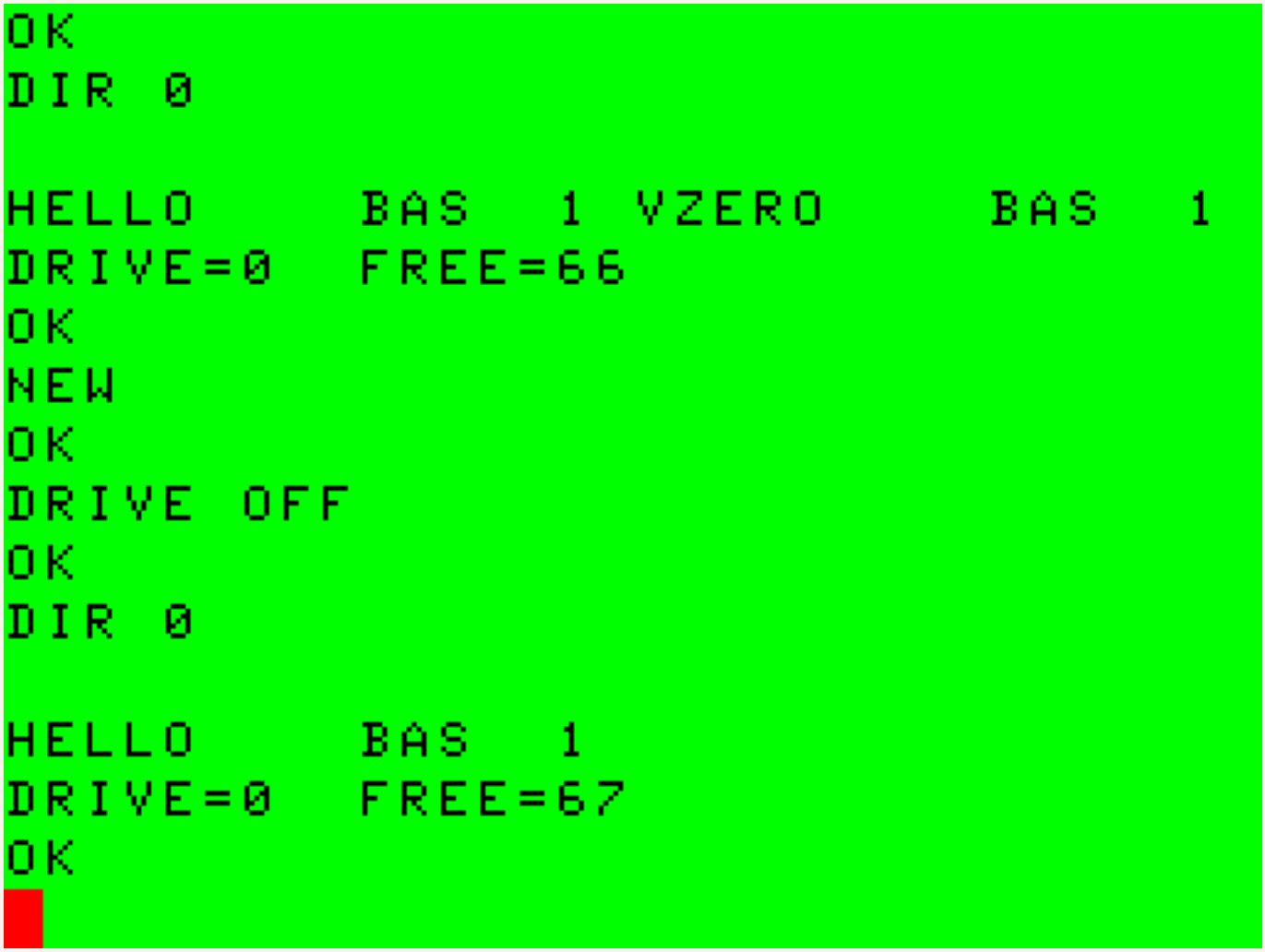


Figure 7: Both RGB-DOS listings identify drive 0. The one before DRIVE OFF contains HELLO and VZERO. The one after it contains only HELLO.

Predict what will happen if you ask the original device to answer again. Then try it:

```
DRIVE ON
LOAD "VZERO:0"
RUN
```

VOLUME ZERO and **OK** confirm that the CoCo read back the marker on the hard disk. You cleared RAM before making that trip. Seeing its message again is a stronger check than seeing the number zero printed next to two directory listings.

If you use your own images later, both copies might contain VZERO, or neither might have HELLO. Check the assigned Mac filenames, the last routing command and the title's instructions. In particular, use **DRIVE OFF** and **DRIVE ON** in this Hard Drive BASIC setup. The standard floppy profile and the SDC profile have different firmware. Read the SDC card and a mounted disk to compare the same drive number there.

Know which part of a hard disk you're opening

From this setup	The name you use	What it reaches
Hard Drive BASIC, just started	Drive 0	RGB-DOS BASIC volume 0 on HD0

From this setup	The name you use	What it reaches
Hard Drive BASIC after DRIVE OFF	Drive 0	The floppy in FD0
Bundled NitrOS-9 system profile	/dd	The OS-9 system filesystem on its main disk
Bundled NitrOS-9 system profile	/h1	The OS-9 filesystem on the second hard disk

The standard Hard Drive BASIC and NitrOS-9 profiles share a *working data image*. In the first profile that image is HD0. In the second it is HD1 because the NitrOS-9 system disk occupies HD0. The shared data image has separate space for BASIC volumes and an OS-9 filesystem. Opening its OS-9 directory shows the OS-9 files in that space. Your fresh personal profile for this lesson has its own working media, so keep it if you want to return to VZERO.

The **New Disk...** command in Virtual Hard Disks asks how much storage to create. A new raw image is empty space until software prepares its filesystem and you save the image to a Mac file. For a bootable setup or a ready-to-use BASIC directory, start with the matching system profile. For a short trip to the OS-9 shell and the two actual directory listings, see Visit the OS-9 side of the disk. To create and recover your own OS-9 file on HD1, follow Leave a captain's log.

The native **Disk Image Editor** also distinguishes the parts of a supported image. **Edit...** beside HD0 or HD1 pauses and locks that machine; the editor's **Volume** picker selects OS-9 or a numbered HDB-DOS Disk BASIC volume. Space totals apply to the selected filesystem. Saving an edited copy retains the whole image; applying to a mounted drive requires an explicit restart. See Edit the files inside an image before using that separate Mac workflow.

On the OS-9 side, you can follow a particular file through its disk sectors.

Record a program you can rewind

Chapter 4 of 8

With a floppy, BASIC can go straight to the directory and find CARGO. A cassette has a beginning, an end and a position somewhere between them. To find CARGO, the CoCo must pass the recorded name on its way along the tape. You can watch it recognize that name.

Put the recorder on your desk

Use **Disk BASIC** for this lesson. Save anything you want from RAM before changing profiles. Load a practice Disk BASIC setup in **Machine → Configuration Profiles…**, resume if paused, and choose **Media → Cassette…**. Clicking the recorder in System view leads to the same controls. Look for the tape name, transport state, motor state and position counter.

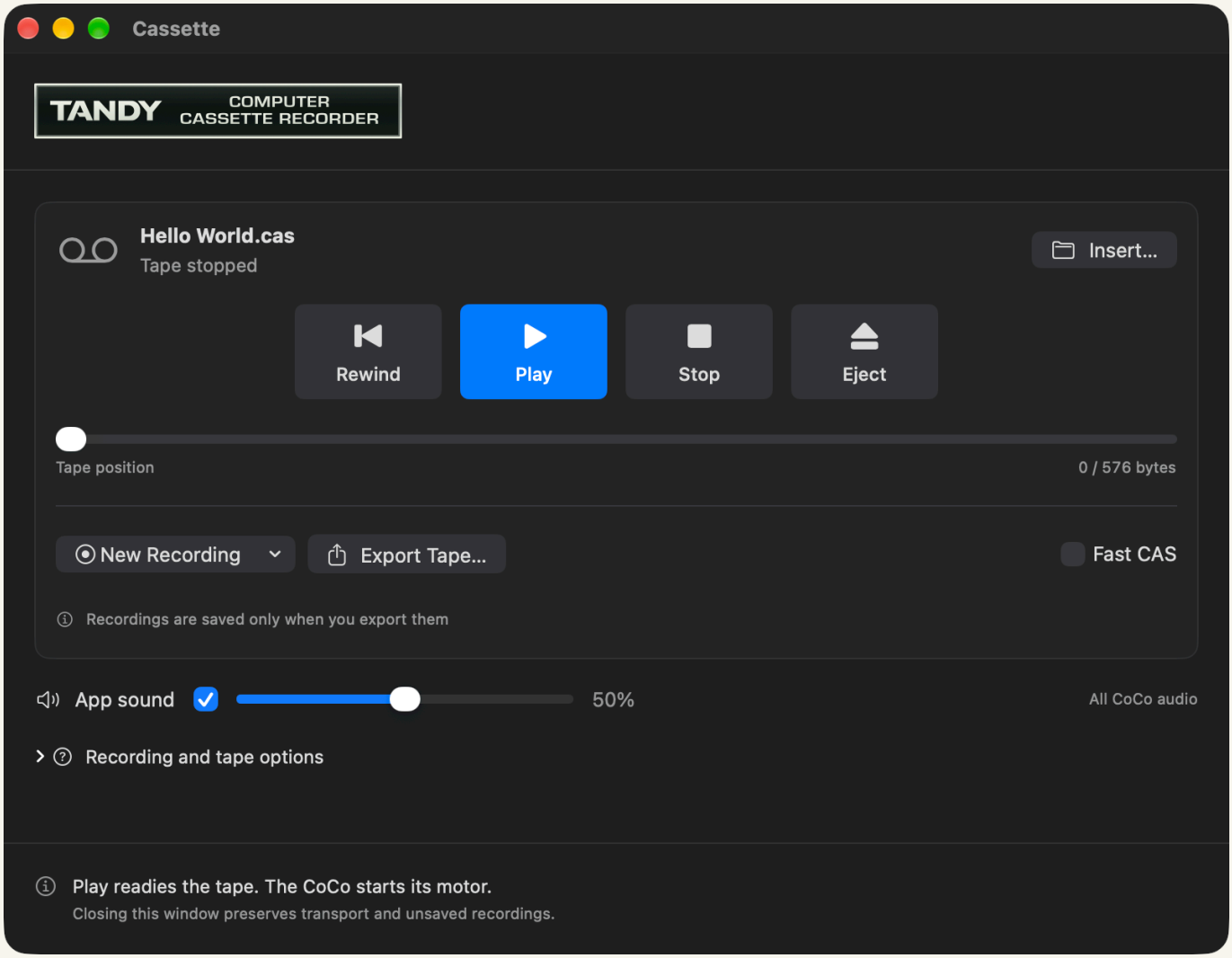


Figure 8: The starter tape Hello World.cas is stopped. New Recording and Export Tape are below the position counter. Use them to make and keep your CARGO tape.

On a CoCo desk the operator pressed buttons on the recorder and BASIC controlled the motor through its cable. Here **Play** gets the tape ready. The CoCo starts its motor when a program asks for input. Seeing that the tape is ready while its counter stands still makes sense at an idle BASIC prompt. The Cassette window can close

while the recorder is in use. Reopen it from **Media** or the tape status when you want to look at the controls again.

Want to try playback first? Save the cargo cassette in your practice folder and jump to Find the recorded name. To make your own, follow the recording steps below.

Record CARGO and keep the tape

1. At a practice BASIC prompt, type `NEW` and press Return to start with empty program memory. Enter the numbered line below. As in the floppy lesson, it stays in RAM until you save it.

```
10 PRINT "CARGO ORIGINAL"
```

2. In **Cassette**, choose **New Recording** → **CAS**. This replaces the inserted tape with a fresh tape in memory and arms the recorder. If an earlier recording needs export, keep it under its own Mac filename before starting another.
3. Return to the CoCo. Type `CSAVE"CARGO"` and press Return. Watch the tape position advance. Wait for **OK** and for the tape motor to stop. The C at the beginning of **CSAVE** sent the program to the cassette interface.
4. In **Cassette**, choose **Stop**, then **Export Tape…**. Give the new file an unused name such as **My Cargo.cas** in your practice folder. During recording you may see **Recording needs export to keep it**. Guest success at **OK** means BASIC finished sending the program to the recorder. Export is the step that keeps the recording in Finder.
5. After the export finishes, choose **Eject**. At the CoCo, enter `NEW`, then `LIST`. Check for **OK** without any numbered lines. You now have an empty BASIC workspace, an empty recorder, and a recording you can insert.

Find the recorded name

1. In **Cassette**, choose **Insert…** and select the CAS file you exported or the practice cassette you saved from the book. Choose **Rewind** to set position to the beginning. Choose **Play**.
2. Focus the CoCo screen and, if needed, resume the machine. Type `CLOAD"CARGO"` and press Return. Watch for **F CARGO**. BASIC found the name you requested. Wait for **OK** before typing another command.
3. Type `RUN` and press Return. Look for **CARGO ORIGINAL** and **OK**. Those words came from the program read from the tape. Its Mac filename can be **My Cargo.cas** even though the program name BASIC looked for was **CARGO**.



Figure 9: After the tape search, the CoCo displays F CARGO and OK. RUN then prints CARGO ORIGINAL.

If the search seems to go nowhere, check the tape filename, the position counter, and whether the guest is running. A counter already at the end means the recorded name is behind the current position. Return to a BASIC prompt, rewind, choose Play, and start the named load again. If the name was found but the greeting has yet to appear, check for **OK** and type `RUN`. Finding, loading and running are separate events.

Compare a compact tape with a waveform

Repeat the recording and playback steps with **New Recording → WAV**. Give the export a fresh .wav filename. Tape position now counts **samples**, while a CAS tape counts **bytes**. Hearing something through the Mac speaker is optional. The counter, found-file message and BASIC result let you check every step visually.

Both forms carried the same program in our exercise. The CAS file occupied **310 bytes**. One WAV recording of the same line occupied **131,160 bytes**. CAS keeps the encoded tape data compactly. WAV represents the signal as samples over time, so its length can vary with recording time. Use the format that matches the material you have or the destination you plan to use.

The **Fast CAS** switch is available with the transport stopped. It accelerates compatible BASIC ROM loads of CAS files. WAV playback follows the waveform. For a recording you want to pass on, keep the actual exported file and try it on a fresh load as you did here. The first chapter's floppy round trip uses the same proof with a different device.

Choose what goes in the slot

Chapter 5 of 8

A game cartridge could give you a title screen as soon as you switched on a Color Computer. Plug in a disk controller instead and you got a prompt. Both cartridges could contain a ROM: a chip holding instructions the computer could read as soon as power came on. The game ROM held a program. The controller's ROM gave BASIC the words it needed to use a disk.

Read the four slots

Start in your saved **Disk BASIC** practice setup. Save anything you want to keep from guest RAM. Open **Cartridge → Cartridges…**. In the usual Multi-Pak setup, three slots are empty and **Slot 4** says **Floppy controller (FD-502)**. Its drive count refers to FD0 through FD3. All four drives connect to that one controller. In System view, the pictured FD-502 in the slot and the drives in the tower are two different places you can click.

Look at the **Startup slot** selector and the **Live routing** readout. Startup is the physical switch you would set before turning on a CoCo. The two live readings answer different questions. Which slot supplies cartridge ROM bytes now? Which slot is selected for device input and output? At the Disk BASIC prompt the usual answer to all three is slot 4. Write down those three numbers before the next experiment.

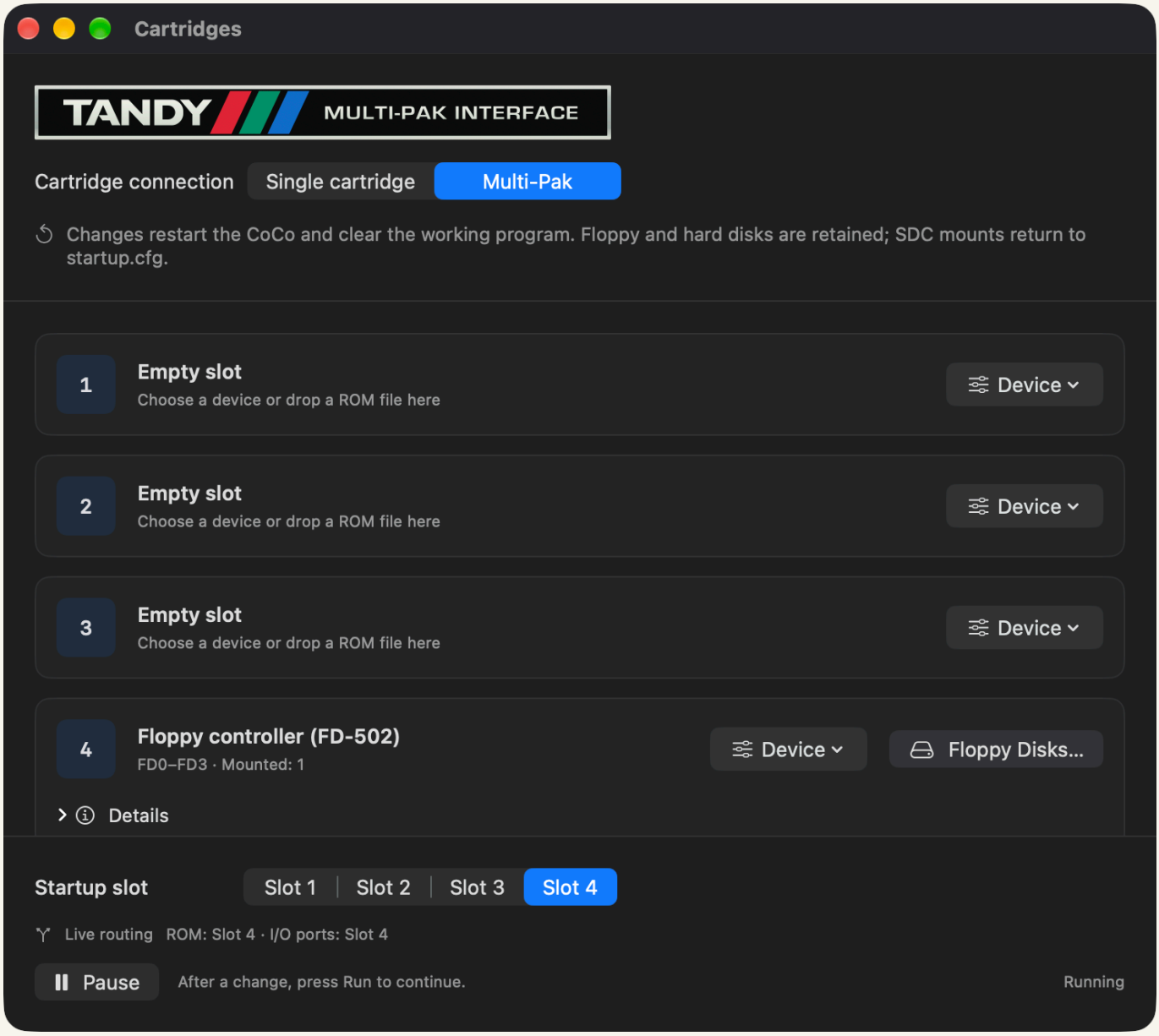


Figure 10: The native Cartridges window shows which slots are occupied and which slot starts the machine.

The **Device** menu on each row lets you assign a cartridge. Read the full menu item before choosing it: if the device is already in another slot, the menu tells you it will move. Selecting a different cartridge, startup slot or connection mode cold-restarts the machine. Resolve any unsaved media first. A cartridge change can leave the machine paused. The Mac toolbar's **Run** resumes it.

A byte in a cartridge

Here is a way to *use* the second live reading. A 512 KB RAM-disk cartridge can hold a byte even while the disk controller's ROM keeps BASIC running. Software talks to this board through numbered addresses. For this experiment, use **Slot 1** → **Device** → **RAM disk (512 KB)** and keep the FD-502 in **Slot 4** with **Startup slot 4**. Confirm the restart. If the illustrated desk gives way to conventional controls, use the Cartridges window. The hardware is still connected.

Resume the guest and click its screen. Finish any previous disk operation before you enter these commands. Enter one line at a time in this exact setup.

```
POKE &HFF7F,&H30
POKE &HFF40,0:POKE &HFF41,0:POKE &HFF42,0
POKE &HFF43,42
PRINT PEEK(&HFF43)
POKE &HFF7F,&H33
```

1. Pause after the first line and look at **Live routing**. The ROM selection stays on slot 4. Device I/O moves to slot 1. The physical startup switch stays where you left it. This lets BASIC run from one slot while reaching the board in another.
2. The second line selects location zero on the board. The third writes the number 42 there. The fourth reads that same location: BASIC should print **42** and return to **OK**. Here `POKE` means write one byte and `PEEK` means read one. The prefix `&H` introduces a hexadecimal number.
3. The last line restores device I/O to slot 4. Do this before trying a disk command, even if you stopped midway. With a BASIC disk mounted in FD0 you can now issue `DIR 0` again.

The RAM disk keeps its contents across a soft or cold reset. Removing its last assignment, loading a profile or quitting loses those contents. Use the owning program's transfer or save feature to copy anything valuable onto persistent media first. A profile remembers the board assignment. The experiment writes a byte on the board; to keep it as a named file takes software that understands the device. Increasing **Machine Settings → RAM** gives the computer more main memory and serves a different purpose.

Try a cartridge that starts a program

If you have a CoCo program cartridge image such as a .ccc file, save your disk setup with **Machine → Configuration Profiles… → Save Current As…**. Save any guest program separately. In a free Multi-Pak slot choose **Device → ROM Cartridge…** and pick the file. Make that slot the **Startup slot**, confirm the restart and resume. The cartridge's own opening screen is the checkpoint. A title that waits at a menu may already be running. Check the program's own instructions for its first key. The Color Scripsit II exercise takes you all the way from its opening menu to typing in the editor.

Single cartridge models a cartridge plugged directly into the computer. Use its single assignment row. **Multi-Pak** gives you four rows and a selector. VCC Plus remembers the two layouts separately. If the System illustration cannot represent the selected hardware, its explanation directs you to the conventional controls. The same file choosers, device details and running guest are available there.

A missing ROM is a different problem from choosing the wrong startup slot. Open the affected slot and read its status. If you moved the file, choose its new location; if its volume was disconnected, reconnect it and retry the unavailable ROMs. See the handbook's cartridge controls. Return to your saved practice profile when finished. Check that **FD-502** is back in slot 4, FD0 has the expected disk, and the BASIC prompt responds. The copy you saved on that disk is ready to load again.

Install, play, and come home

Chapter 6 of 8

A friend hands you a disk image. A Library search turns up a tape and a disk with the same game title. In either case, someone has to choose a machine, insert the right medium and start a particular program. You can do those jobs yourself. For many catalog entries, the Library can do all three.

Launch a file you already have

Try the reader's side of a manual setup first. Save your work and start at the **Disk BASIC** prompt. In the book, select the CARGO practice disk and save a copy in a Finder folder of your own. In a browser, use the download link or copy CARGO.DSK from this book's **media** folder.

1. Open **Media → Floppy Disks…**. At FD0 choose **Insert…** or **Replace…** and pick your copy. You can also drop a file on the appropriate drive row. Read the name beside FD0. Protect the disk for this reading visit.
2. Resume the machine if it paused. Focus the CoCo display, enter `DIR 0`, and find **CARGO BAS**. The image is mounted. Its directory lists a program that BASIC can run.
3. Enter `LOAD "CARGO:0"`, then `RUN`. **CARGO ORIGINAL** tells you which file you actually ran. After **OK**, you can keep working or eject the disk. For a file you have used before, **File → Open Recent Media** remembers earlier choices.

For a CoCo block-format machine-language binary, **File → Load Binary…** loads its blocks and starts at the file's execution address. For a disk image containing a binary, insert the *disk* and follow the title's loading instructions instead. **File → Open Media…** also accepts several media types. Check the selected drive, slot or recorder before confirming. The file guide collects the common choices in one place.

Choose an edition of Bedlam

Open **Media → Software Library…**, or click the wooden shelf in System view. Choose **All Software**, clear any old filters and search for **Bedlam**. Choose the Tandy adventure and look for its **disk** and **cassette** editions. The catalog follows the Color Computer Archive's collection. Its edition and source details help you identify what was preserved there. A title can appear on several media, and some similarly named releases may have different publishers or different loading procedures.

Under **Using this software**, read the recommended machine and loading instructions for your selected edition. Then read **Compatibility** for known requirements. Think of the difference this way: one tells you which disk to insert and what to type; the other may tell you which processor, RAM or controller you will need. The heart saves the title to **Favorites**. Your favorites can include software you have never installed.

To find a game's keys, look for **Read Manual** or **Read Instructions**. A PDF opens in the app's document reader; the instructions button opens a text document. The Library may need a connection for the first cover, manual or software download; it caches material it has retrieved. This book and its CARGO exercise work offline. For more browsing controls, see Choose a version that fits.

Compare Install Only, Play and Set Up

On a connected Mac, choose one Bedlam edition for an optional visit. **Install Only** checks the download and makes local working files. Watch the installed status appear in the Library. Look back at the running CoCo: the program you left there can still be running. **Show Files in Finder** shows the files available for use on this Mac. Writable media here belongs to your installation; the Library keeps the original package separately.

Save anything you want from guest RAM before starting the edition. Choose **Play…** for an installed edition with a launch sequence. If you started with **Download & Play…**, the install leads into the same setup preview. At **Start Software**, look for the processor, RAM, signal and the item the Library will mount or plug in. Check the destination before you choose **Start**. If the collection has several possible starting files, use **Choose Starting**

Media and the edition's instructions. **Cancel** at the preview lets you investigate further without restarting the CoCo.

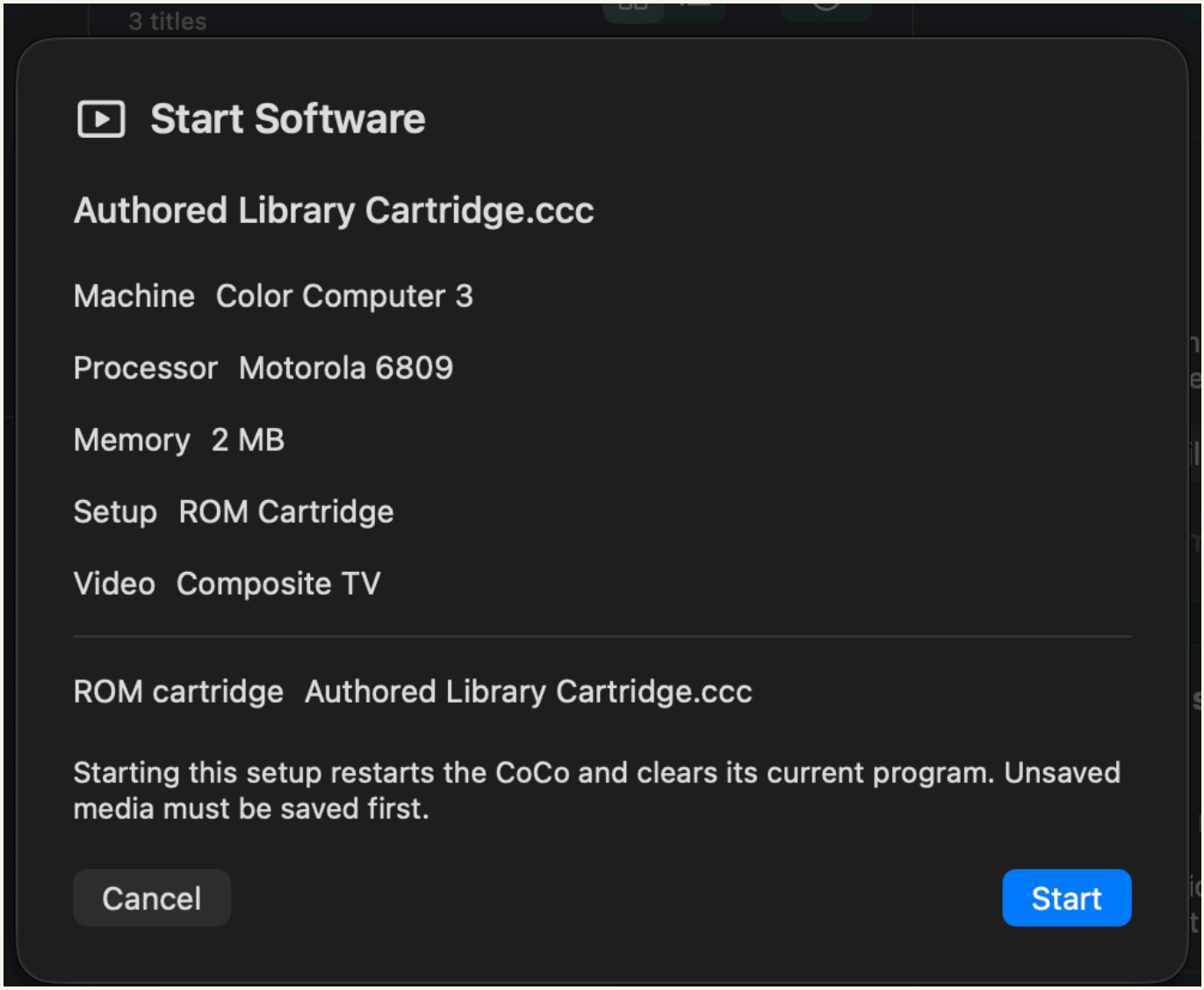


Figure 11: The Start Software sheet for an example cartridge identifies the processor, memory, video signal and cartridge file before asking to restart.

Bedlam gives you a useful comparison: the disk edition reaches the adventure by reading FD0, while the cassette edition reads the tape. Both reach the opening passage about waking from a bad dream. The media indicators and filenames can differ even though the story looks familiar. Click the guest screen when the story asks you to type.

Some editions offer **Download & Set Up…** or **Set Up…**. This is a manual launch. Read the **Manual loading** part of the preview and choose **Set Up** when ready. The Library prepares the equipment and media; the instructions tell you how to continue from the guest. For a blank data disk, reaching a prompt with the disk available can be the entire job. For CARGO you already know how to complete the two remaining steps yourself.

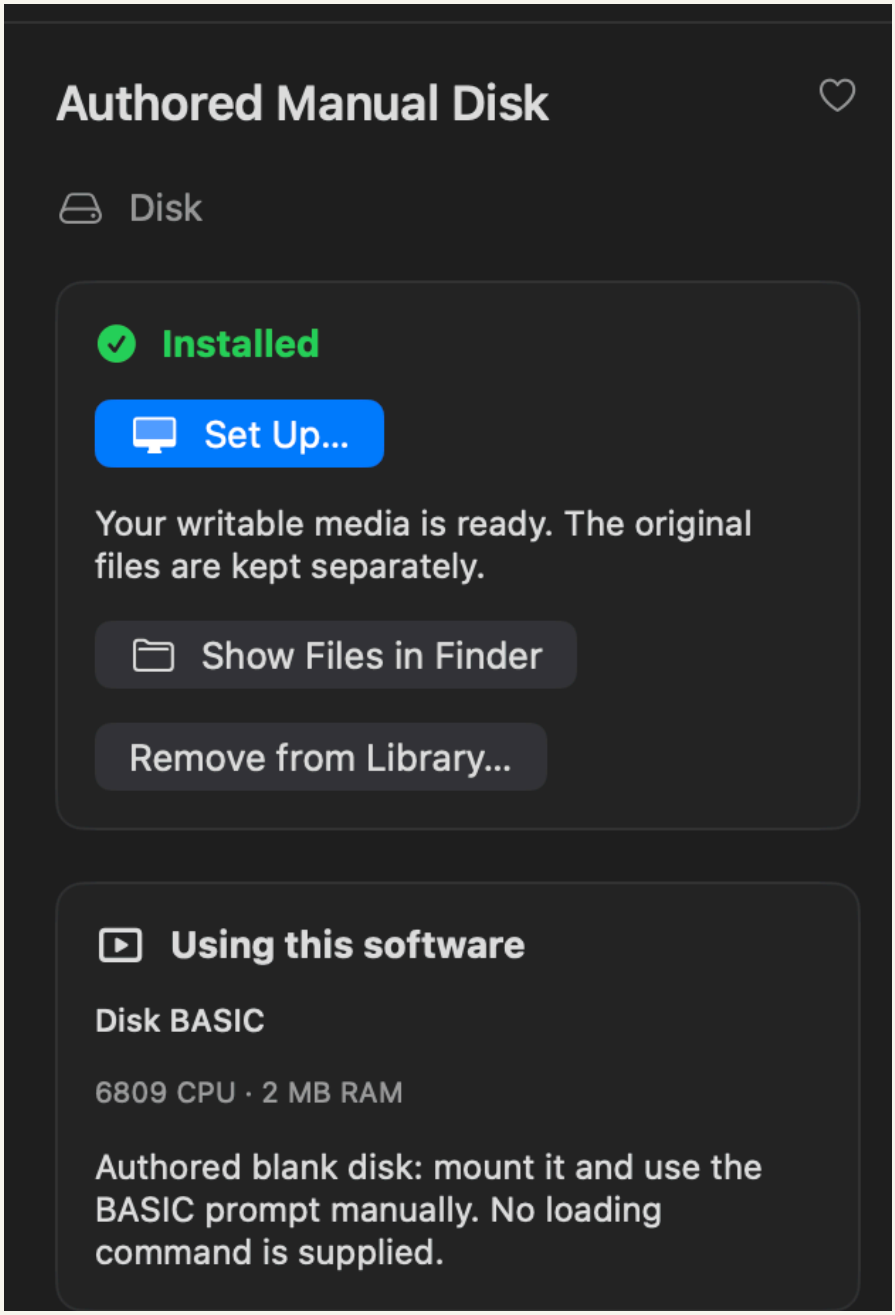


Figure 12: In the detail panel for an installed example disk, Set Up and Show Files in Finder appear above its manual loading guidance.

Return with your work

If you save a game, save it **inside the guest** on a writable working disk using the game's own instructions. Give the disk time to finish saving. A saved score can be part of a file inside the image without appearing as a separate Mac document. A write-protected disk, an unexported recording or a game that never writes a save needs a different plan before you leave.

Open **Media → Software Library...**. Near the top of the window, choose the curved back arrow **Return to Previous Setup...**, then **Restore Setup**. This option appears when the Library was able to capture your earlier configuration. It restarts the machine with the previous equipment and media assignments. A prompt where a game used to be is normal. Reload a program from the disk where you saved it. A visit you arranged yourself through the drive window can be undone by loading your named practice profile.

Return to **Installed** to find the game's working files again. **Remove from Library…** removes its installation entry while preserving its files on the Mac. A later installation gets a fresh working copy. If you want to keep playing from a saved game, locate your previous copy before starting over. Checking both the filename on the Mac and the guest's directory is a skill you can use outside the Library, too.

Send a receipt, a tune, or a voice

Chapter 7 of 8

Where would you look for a message that BASIC printed, but that never appeared on the screen? On a real CoCo you might have heard a printer start up at the other end of the serial cable. In VCC Plus you can catch that same stream of characters in a window. With another piece of equipment in a cartridge slot, the computer can also play a score or say a sentence out loud.

Print a receipt for CARGO

Start at **OK** in **Disk BASIC**, with the tape stopped. Open **Media → Printer Output...**, or click the little printer behind the cassette recorder. If the window already contains a print job you want to keep, stop capture and save it first. Choose **Clear...** and confirm when you are ready for an empty window. Click **Start Capture**. Return to the CoCo display and type this line, followed by Return.

```
PRINT #-2,"CARGO RECEIVED"
```

The CoCo returns to **OK**. Look in **Printer Output** for **CARGO RECEIVED** and **15 bytes**. Count the five letters of CARGO, one space and eight letters of RECEIVED. That makes fourteen. The last byte is a carriage return. On paper it tells the printer that the line has ended; the Mac text preview displays a new line. The `#-2` in BASIC's command chose the printer as its destination.

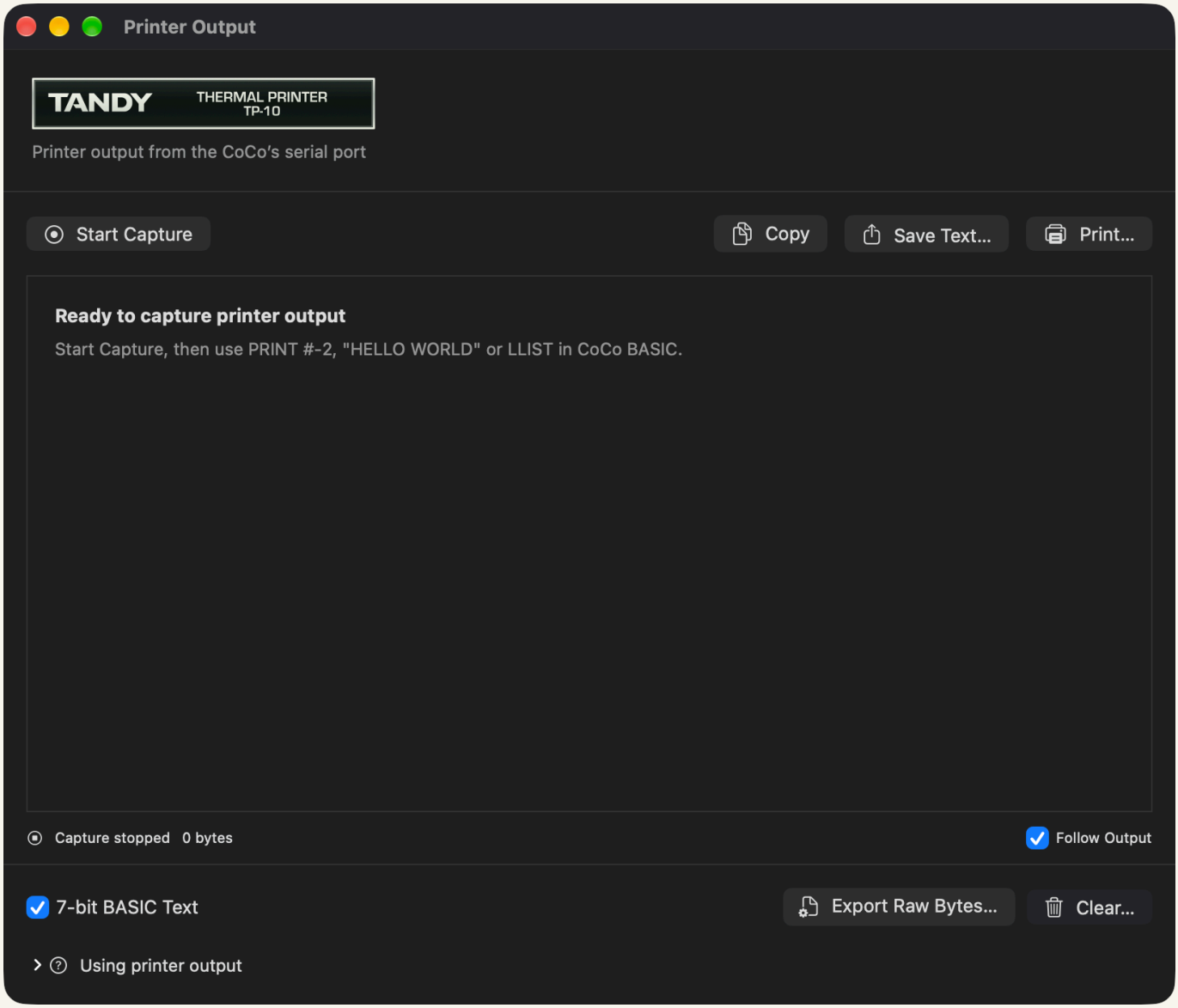


Figure 13: The native Printer Output window places capture and save controls above the text and a byte count below.

Click **Stop Capture**, choose **Save Text...**, and open your saved receipt in a Mac text editor. To keep the exact fourteen character codes *and* the end-of-line byte, choose **Export Raw Bytes...**. The **7-bit BASIC Text** switch affects the text interpretation. A printer control code may appear as a marker in the text preview. Raw export keeps the code itself, useful when examining how an older program drove a printer.

You can close the window during a print job and open it again. Capture continues until you stop it. **Clear...** stops capture and discards the collected output after confirmation. Stop capture between jobs: other software can use the same output register, and a long-running capture can fill up. **Print...** opens the ordinary Mac print dialog if you would like a paper receipt.

Score and play William Tell

Orchestra-90 contains stereo sound hardware. Its optional program ROM turns the CoCo into a music editor with a score already in memory. If you have that ROM, you can go from a quiet desk to Rossini's *William Tell Overture* in a few keystrokes.

1. Save anything important from guest RAM and note your practice profile. Open **Cartridge → Cartridges...**. Choose **Orchestra-90** as the device in a free Multi-Pak slot. In that slot's details confirm the 8 KiB music

program ROM, or choose yours. Set **Startup slot** to that slot, confirm the restart and resume the machine if paused.

2. At the cartridge's opening screen, give the guest keyboard focus and tap **Space**. Look for **WILLIAM TELL OVERTURE** and **BY ROSSINI**. Those two lines are comments identifying the score the synthesizer will use.
3. Type **S** and press Return. Give the cartridge time to score the music. Then type **P** and press Return to play it. Start at a comfortable volume. The left and right channels carry distinct parts. **Esc** sends the CoCo's Break key if you want to stop.
4. If you can see the score but hear silence, confirm the machine is running at normal speed, app sound is enabled, and the Mac output volume is up. If you return to Disk BASIC immediately after restarting, check the music ROM and its startup slot. A program can use the stereo ports without the music ROM; this particular lesson uses the program in that ROM.



Figure 14: The real Orchestra-90 editor displays WILLIAM TELL OVERTURE and BY ROSSINI after the opening screen.

Have the computer introduce itself

The Speech/Sound Cartridge takes another approach. Leave your FD-502 in startup slot 4 so BASIC boots as usual. The speech board waits for a program to send it characters. In this exercise, you can watch BASIC send a short message and hear the cartridge read it.

1. Return to your saved **Disk BASIC** practice profile. Open **Cartridge → Cartridges…** and assign **Speech/Sound Cartridge** to free slot 2. Keep **Startup slot 4**. Expand the speech board's details. It needs both its internal **PIC firmware** and its **Speech ROM**. If either is missing, use **Choose PIC Firmware…** for pic-7040-510.bin and **Choose Speech ROM…** for sp0256-al2.bin. Confirm any restart and resume at **OK**.

2. Open VOICE.bas in BASIC Development. If reading in a browser, save the file first and open it in the BASIC workspace. Save anything you want to keep from guest RAM, then choose **Run** in that workspace.
3. Look for **SENDING GREETING**, then **GREETING SENT**. The voice says "Hello from your Color Computer." The second screen message means the last character made it through; the voice can still be finishing the sentence. **CHECK SPEECH CARTRIDGE** means the program timed out waiting for the board to accept another byte. Check the assignment and both ROMs before running it again.

```
10 REM A GREETING FOR THE SPEECH/SOUND CARTRIDGE
20 A=&HFF7E
30 M$="HELLO FROM YOUR COLOR COMPUTER. "+CHR$(13)
40 PRINT "SENDING GREETING"
50 FOR I=1 TO LEN(M$)
60 T=0
70 IF PEEK(A)>=128 THEN 100
80 T=T+1:IF T<500 THEN 70
90 PRINT "CHECK SPEECH CARTRIDGE":END
100 POKE A,ASC(MID$(M$,I,1))
110 NEXT I
120 PRINT "GREETING SENT"
```

Follow one letter through the loop. Line 70 checks whether the cartridge is ready for it. Line 100 converts the letter to a character code and sends it to the board. The carriage return at the end of the string closes the message. To try another greeting, change the words inside the quotes on line 30. Keep the space and the carriage return expression at the end.

When finished, reload your saved practice configuration and check the controller and FD0 assignment. Your receipt is a Mac file, VOICE.bas is editable source, and the music you heard came from the score in cartridge memory. Each can outlast a listening session in a different way.

Leave the original just as you found it

Chapter 8 of 8

Someone hands you a disk and says, “Try changing the greeting, but let me have my original back.” When you finish, both disks will have a BASIC program called CARGO. Looking at the two directories will tell you almost nothing. What could you do to prove which disk has the change?

Make two practice disks

Start with the CARGO.DSK practice image or the disk you made in Put a program somewhere. In Finder, make two copies in a practice folder. Name one **Cargo Original.dsk** and the other **Cargo Working.dsk**. Put the supplied practice file away. You can repeat this experiment as often as you like.

Save any other work from the CoCo. Choose **Disk BASIC** in **Machine → Configuration Profiles…** and load the setup. Open **Media → Floppy Disks…** and put *Cargo Original.dsk* in FD0. The button says **Replace…** if a disk is already there. In FD0’s **Disk options**, turn on **Write protected**. Read back the filename and the words **Write protected** together. On a narrow screen, scroll inside the Floppy Disks window to see the options.

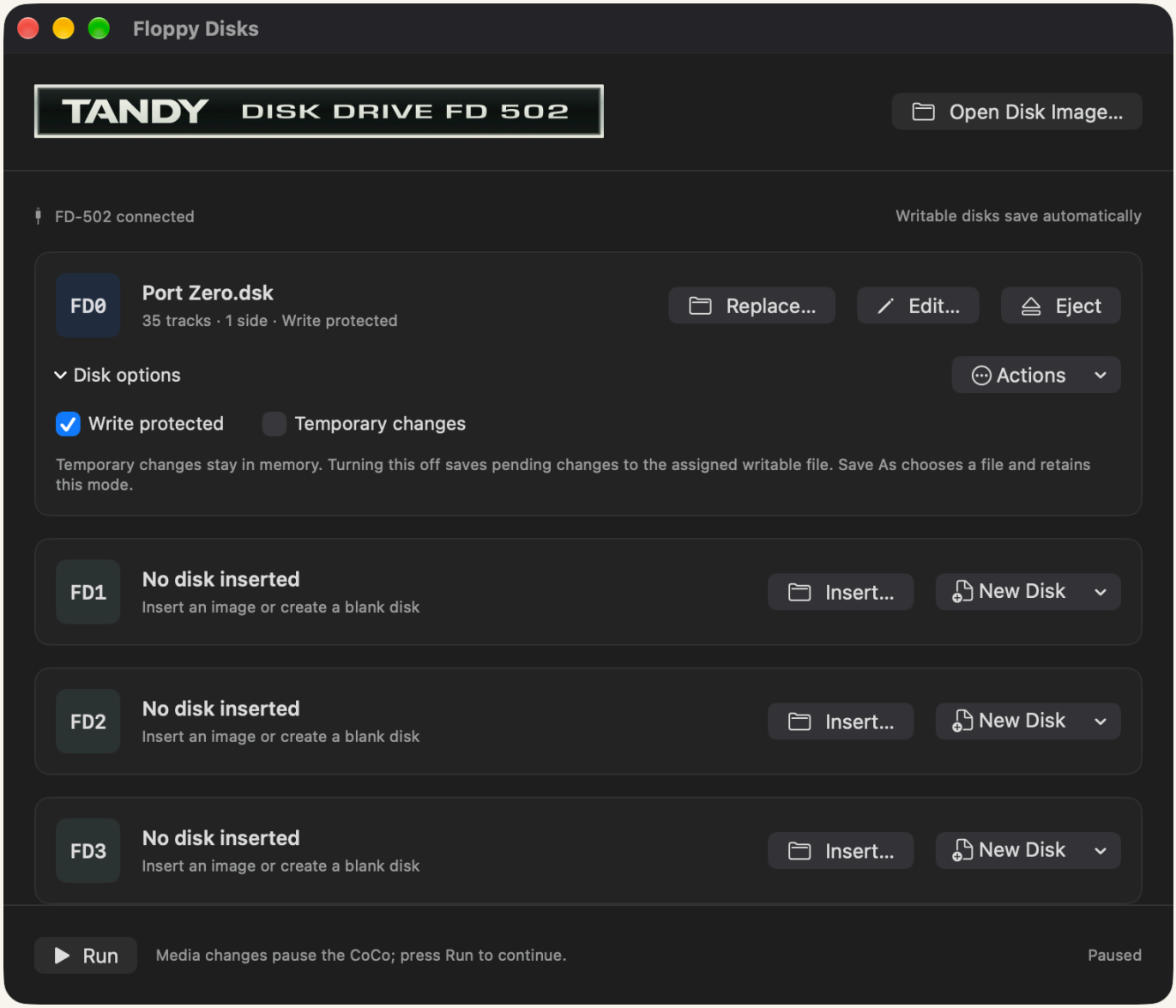


Figure 15: FD0 shows Write protected with its matching box selected. This representative practice disk is Port Zero.dsk; your lesson copy should be Cargo Original.dsk.

Media changes pause the CoCo. Click **Run** in the window footer to resume, then click the CoCo display to type. At the BASIC prompt, enter each of these commands and press Return:

```
LOAD"CARGO"
RUN
```

You should see **CARGO ORIGINAL**, then **OK**. The disk supplied the program, and BASIC ran it in RAM.

Change the greeting in RAM

Enter these two lines:

```
10 PRINT "CARGO COPY"
SAVE"CARGO"
```

Expect **?WP ERROR** from BASIC. WP means *write protected*. Line 10 changed as soon as you pressed Return after typing it. The error came on the next command, when BASIC tried to put that line on disk. Leave the program in RAM while you swap disks.

Open **Media → Floppy Disks…** and use **Replace…** on FD0. Choose **Cargo Working.dsk**. Read the filename again. For *this working disk*, leave **Write protected** and **Temporary changes** off and check that the row says **Saves automatically**. Resume the machine. At the guest prompt enter:

```
SAVE"CARGO"
```

This time BASIC should say **OK**. Let the Mac finish saving the disk. The directory entry still says CARGO. What will the program actually say the next time you load it?

Prove which copy changed

1. With the working disk saved, cold-restart the practice machine. A cold restart clears the program in RAM. Check that **Cargo Working.dsk** is still in FD0, or insert it again. Resume if paused.
2. Type `LOAD"CARGO"`, then `RUN`. Look for **CARGO COPY** and the **OK** prompt.
3. Replace FD0 with **Cargo Original.dsk** and select **Write protected** again. Resume and enter `NEW`. This clears the stored line you just ran. Now type `LOAD"CARGO"`, then `RUN`. Expect **CARGO ORIGINAL** and **OK**.

Disk file attached on the Mac	Name requested by BASIC	What came back
Cargo Working.dsk	CARGO	CARGO COPY
Cargo Original.dsk	CARGO	CARGO ORIGINAL

That last load matters. If you simply swapped the two disks and typed RUN again, the CoCo would run the version already sitting in RAM. Loading each program after clearing RAM lets the disks answer the question.

Keep a change only if you like it

Some games need to write to a disk before they will run. Sometimes you just want to try a change and decide later whether to keep it. Make a fresh Finder copy of **Cargo Original.dsk** and call it **Cargo Trial.dsk**. Use that disposable file for this variation.

1. Mount the trial in FD0. With the disk clean, turn on **Temporary changes**. Allow guest writes by clearing **Write protected** if selected. Resume the machine.
2. Load CARGO, enter `10 PRINT "CARGO COPY"`, and enter `SAVE"CARGO"`. This time BASIC accepts the write. Open **Floppy Disks…**. FD0 says **Unsaved changes · Save As to keep them**.
3. Choose **Save As…** on the FD0 row. In the Mac save dialog choose your practice folder and enter a different filename, such as **Cargo Working Export.dsk**. Read the folder and filename before you save. After saving, check which filename appears on FD0. It is now the exported copy, and its save mode is still **Temporary changes**.
4. Finish with the same proof. Start a fresh practice session. Load and run CARGO from the exported disk, then from the unchanged trial. The export should print **CARGO COPY**, and the trial should print **CARGO ORIGINAL**.

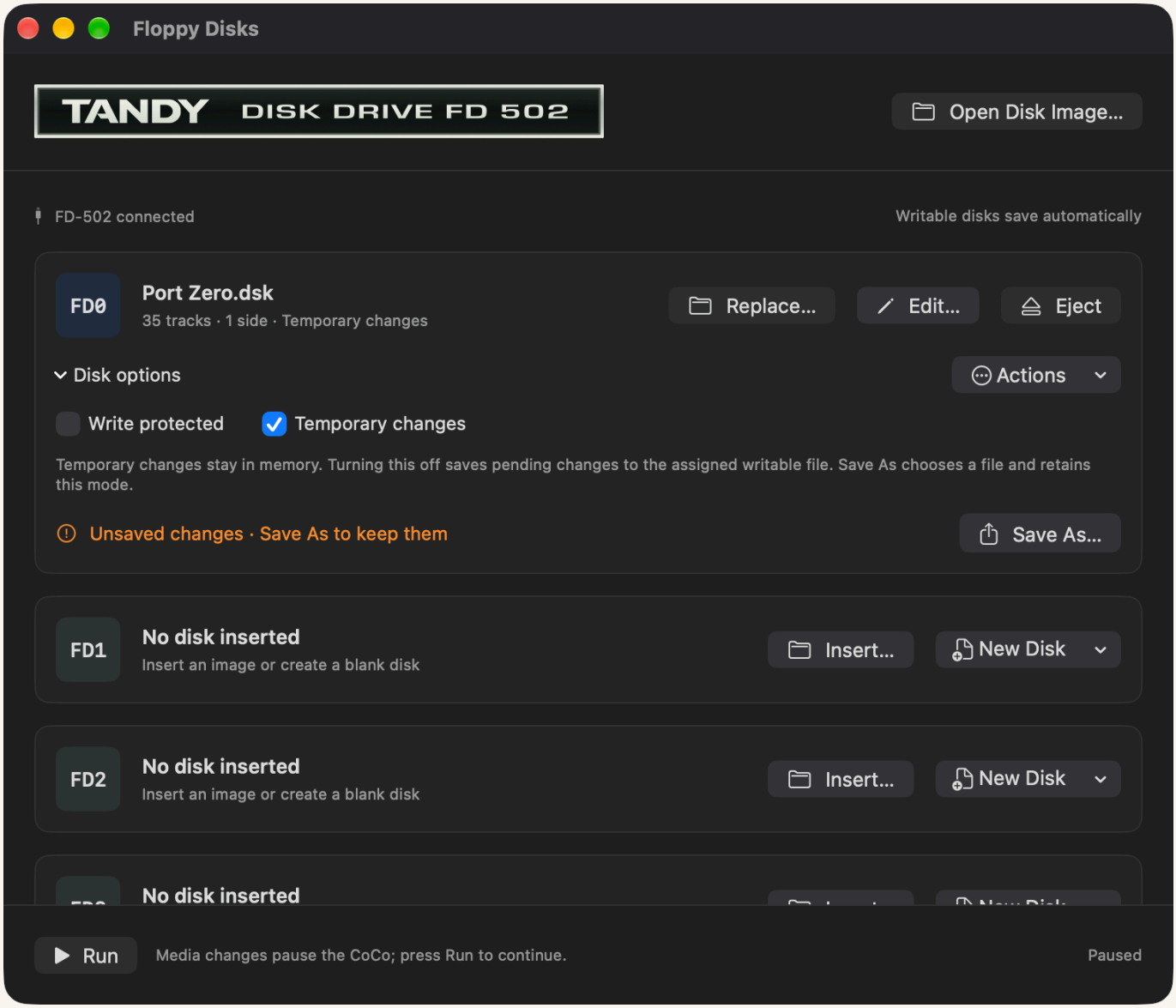


Figure 16: After a guest SAVE, FD0 shows Temporary changes and offers Save As to keep them. This representative Port Zero.dsk rehearsal saved CAPTEST; the lesson above saves CARGO.

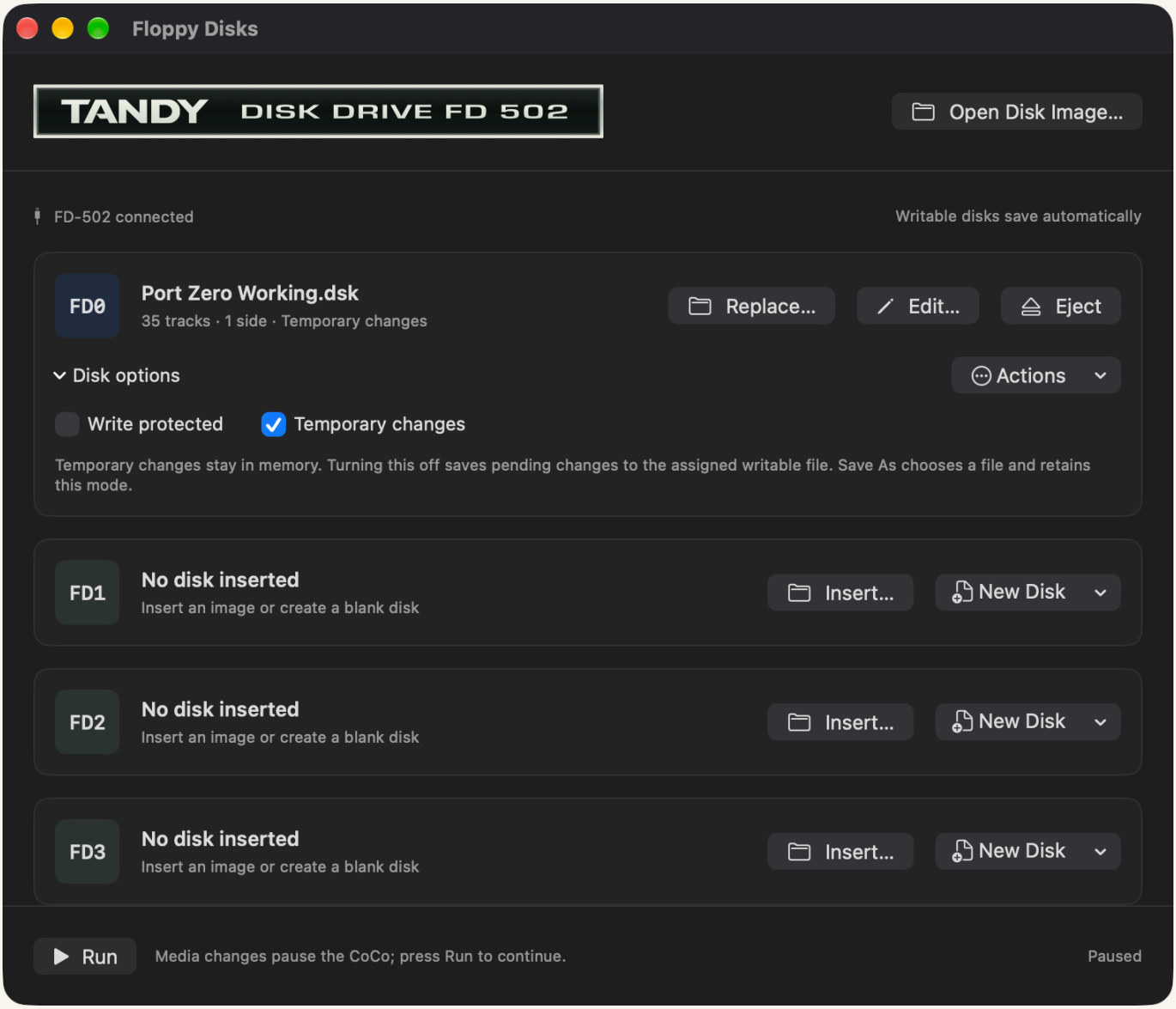


Figure 17: After Save As, FD0 names the new Port Zero Working.dsk copy. The warning is gone and Temporary changes remains selected. Your lesson export should show Cargo Working Export.dsk.

Turning **Temporary changes** off while edits are pending allows those edits to save to the assigned writable file. To abandon a temporary experiment, choose **Actions** → **Discard Changes and Eject...** and review the confirmation. Discard gives up the pending changes. It cannot roll back writes that already reached a file while automatic saving was on.

When the Mac cannot save

The CoCo can accept a write just before VCC Plus discovers a problem with the Mac file. The app pauses and the affected drive row explains what happened. Keep that disk mounted while you decide what to do. If you have the only copy of your work in memory, **Save As...** to a fresh filename on a writable volume preserves the current image.

What you see	What to check	How to proceed
BASIC says **?WP ERROR**	Read the protection state on the drive BASIC used. Image-header or Mac file permissions can enforce protection.	Load a personal writable working copy, then try the guest save again.
FD0 says **Unsaved changes · Save As to keep them**	Check whether temporary mode is on or the image still has no backing file.	Save As under the name you want to keep. Check that FD0 now displays it.
A previously attached disk is missing when you reopen the app	Check whether its external volume is mounted or the Mac file moved.	Reconnect the volume and choose **Retry Remembered Disks**. Use **Replace…** to choose a file that moved.
A save failed after the Mac volume disconnected	The current guest edits may still be in the drive.	Reconnect the original volume and use **Retry Save** if its file is still the expected version. Use **Save As…** elsewhere if you need to rescue the image first.
Another app changed the disk file	You have two versions worth keeping: the changed file on the Mac and the guest's version in memory.	Save As to a different filename. Compare disposable copies later. Avoid mounting the same writable file in two emulators at once.

You can now hand both copies back. You know what each contains because you made the machine read them, and you can tell which save belonged to BASIC and which belonged to macOS. Keep the User's Handbook's recovery chapter handy for missing media and configuration problems beyond this exercise.

What have I got?

Appendix A

Start here if someone has handed you a folder of CoCo files. Pick one file and look at its name in Finder. If macOS hides its ending, use **File → Get Info → Name & Extension**. Keep the folder together. A readme beside a disk can tell you which machine to use, which disk to insert first, and which key starts the program.

Pick up the practice files

Make a **Cargo Practice** folder in Finder. Choose a link below. In the app reader, a disk or cassette opens a save panel so you can keep your own copy. From a web preview, move the downloaded file into Cargo Practice before attaching it. If you opened the HTML directly from Finder, you can copy the file from the book's **media** folder in Finder. A BASIC source link opens an editable example in the native reader. In a browser you can save the source and open it in **Develop → BASIC Development...**.

File	What you'll find	Where to start
BLANK.DSK	An empty floppy with a Disk BASIC directory already prepared. Save your copy as **CARGO.DSK** for the first exercise.	Make your own cargo disk
CARGO.DSK	The finished floppy. Its directory contains one BASIC program called CARGO. Use your own copy of this image to join the course at the SD card or working-copy exercise.	Carry a disk to an SD card
CARGO.CAS	A cassette with a recording of that same program under the name CARGO.	Find the recording
CARGO.bas	One line you can edit or type at the BASIC prompt. It prints **CARGO ORIGINAL**.	Follow a program from RAM to disk
VOICE.bas	A BASIC program for the Speech/Sound cartridge. The output chapter explains the required firmware and the result to expect.	Try the sound cartridges

If you've already made your own CARGO.DSK, give the supplied finished disk a different Finder folder. That lets you compare your work without replacing anything.

Which device opens this file?

You found	Look for	First move in VCC Plus
A .dsk or .jvc floppy image	A readme or catalog entry naming Disk BASIC, another operating system, or a particular controller. The filename's ending alone does not tell you which filesystem it contains.	In **Floppy Disks...**, mount a copy in the drive named by the program, usually FD0. Enable **Write protected** for the first look. A Disk BASIC directory is useful evidence on a BASIC disk.
A .dmk or .vdk floppy	A track image or a disk with a header. DMK can carry information about individual tracks that a simple sector image omits.	Start with the FD-502. It accepts supported DMK and uncompressed VDK images. **Save As...** preserves their representation. Renaming a file to .dsk does not convert its contents.
A folder containing several disk images, perhaps with startup.cfg	Directions that name CoCo SDC or SDC-DOS. A program may expect later disks under their original names.	Choose a **copy of the folder** as the card in the CoCo SDC slot. Keep it protected while exploring. The SDC accepts ordinary flat sector images

You found	Look for	First move in VCC Plus
		and a narrower range of headers than the FD-502. Choose the controller the author tested.
A .vhd, .os9 or .img file	Notes identifying a virtual hard disk or a floppy. Both the device and the filesystem matter. An .os9 file can be a floppy image.	Use **Virtual Hard Disks…** for a documented hard disk and **Floppy Disks…** for a supported floppy. Read the drive-zero map before writing.
A .cas or .wav tape	A title and a loading command in its instructions. WAV can also be ordinary audio.	Use **Cassette…** → **Insert…** on a CoCo tape image. Rewind, choose Play and use the guest loading command for that recording.
A .ccc or cartridge ROM	A cartridge label or release notes. Firmware for a controller can have a .rom or .bin ending too.	For a program cartridge, use **Cartridges…**, choose a ROM cartridge and its startup slot. For controller firmware, use the named device's own ROM control.
A .bin program	An author's instruction to use LOADM and EXEC, or a description of a CoCo loadable binary.	A supported block-format binary can use **File** → **Load Binary…** in the expected machine setup. Save the current program in RAM first. For a binary already inside a disk image, follow the guest instructions for that disk.
A .bas listing or a ZIP download	Is the .bas readable text? Does the archive contain a readme, several files or some combination of formats?	Open plain-text BASIC source in **BASIC Development…**. A tokenized file inside a CoCo disk belongs to the guest. For a Library edition, use the installer and read the selected edition's instructions. For a personal ZIP, unpack a copy and examine the contents.

What is the machine telling me?

Opening a floppy-compatible file in Finder or **File** → **Open Media…** now asks whether to **Edit Disk Image** or **Insert into Floppy Drive**. Editing is the default for a general open. For insertion, select FD0–FD3 and **Keep current state**, **Pause** or **Run**. Run resumes only after insertion succeeds; it does not type a boot command or start a program from the disk. An OS-9 file can also retain its hard-disk insertion choice. Use the explicitly named device in the table above when you already know where the image belongs. Inspect disk files from the Mac explains the editor's separate save and restart rules.

You see	Check one thing first	What to do with the answer
A familiar directory, but your new program is missing	Which profile and which drive answered?	Under RGB-DOS, compare the floppy and hard-disk routes. Under SDC-DOS, check the mounted image. Use the VZERO experiment to tell two directories apart.
BASIC says **?WP ERROR**	Open the row for the drive you actually used. Does it say **Write protected**?	Leave protection on for a treasured disk. Write the program still in RAM to

You see	Check one thing first	What to do with the answer
		a separate writable copy. Prove which copy changed.
BASIC returned to **OK** but the Mac says **Unsaved changes**	Is the row set to **Temporary changes**, or is it a new disk with no backing file?	Choose **Save As...** and use a distinct filename. Check which filename the drive row shows afterward.
The Mac reports a failed save	Is the backing volume disconnected, read-only, or changed by another program?	Reconnect a missing volume and use **Retry Save**. If another copy changed, rescue the pending guest edits with **Save As...** under a new name. Keep both until you compare them.
The disk is mounted, but typing does nothing	Does the Mac toolbar offer **Run**? Is the CoCo display focused?	Resume a paused machine and choose **Input → Focus CoCo Display**. Then give the guest its loading command.
BASIC keeps looking for a tape file	Is the right tape inserted, rewound and set to Play?	Cue it again and check the guest filename. A Mac filename and the name recorded on the tape can differ.
A title starts at an unexpected prompt	Which slot supplied the startup ROM? Does the selected edition require a manual loading step?	Read the title's instructions. Check the cartridge assignment before changing RAM or installing another ROM.
A newly attached sound cartridge stays quiet	Does its slot show the required firmware? Is sound muted, or is the machine paused?	Try the known sound exercise. A sound device may be waiting for a command from a program.
An image is unavailable after moving it in Finder	Which file path does the app remember?	Put it back and retry, or choose the file in its new location. A saved profile keeps file locations. It cannot reconstruct a missing file.

For a supported OS-9 image, the editor also offers Move To… and folder drops for regular files inside the same disk. This does not add folders to Disk BASIC or transfer a file between two disk images.

Beside the original hardware

Appendix B

A disk drive, a floppy controller and a BASIC disk command are three inventions working together. Owners of the original machine could point to the drive and the cartridge. The command they typed chose a file on the disk inside. You can make the same distinction in VCC Plus, even when a single Mac file stands in for a whole disk.

Historical reading

- *Color Computer Disk BASIC Quick Reference Guide*. InterTAN Canada Ltd., 1987. Disk commands, pp. 3–8, and errors, p. 9. Compare the directory, save and load entries with the CARGO floppy exercise.
- Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986. Commands, pp. 2–17. Useful for the tape and printer statements as well as for the BASIC program recorded in both formats.
- Tandy Corporation. *TRS-80 Multi-Pak Interface Owner's Manual*. 1983. The equipment setup and slot selector explain the hardware behind the cartridge chapter.
- Lindner, Tim, producer. *CoCo SDC*. Fourth version, February 2023. Hardware and software by Darren Atkinson. Read the card and virtual drive descriptions alongside the two directories in the CARGO card exercise.
- Radio Shack. *TRS-80 TP-10 Operation Manual*. Catalog no. 26-1261. Introduction and setup. The physical printer used thermal paper. VCC Plus captures the characters that would travel down its serial connection.
- Tandy Corporation. *Orchestra-90 CC Stereo Music Synthesizer with Percussion for Tandy Color Computer*. Special Composer's Edition, catalog no. 26-3143. The mini-concert on pp. 8–10 describes scoring and playing the ROM demonstration.
- Tandy Corporation. *Speech/Sound Cartridge Owner's Manual*. 1984. The speech and sound programming sections explain how a program commands the cartridge.

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

Pick your next experiment

If saving and reloading CARGO made you curious about what's inside a *data* file, Keep a flight log shows how a BASIC program writes and reads records. To trace the same BASIC program one statement at a time, try Find the leak. For the guest shell that sees directories such as /dd and /h1, Visit the OS-9 side of the disk is a short tour you can take now.

For a Mac setting, go to the User's Handbook control index. In the C volume, follow a word from FD0 into guest RAM. A compiled program opens a separate file and reads its data. Keep the distinction between an image and a file inside it. Both languages rely on it.

About the pictures and files

The cover uses the VCC Plus cassette recorder illustration. The window pictures show the working Mac application, and guest screens show output from the emulator. A few pictures show another practice filename or a Library title so you can recognize a control across different setups. Captions identify those examples. Software titles, manuals, cover images and trademarks retain their respective credits. See the artwork notice.

The practice files belong with the lessons. Keep an original set and put your own edits on copies. You can hand someone the resulting disk or cassette and ask them to find out which program you left there.

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.

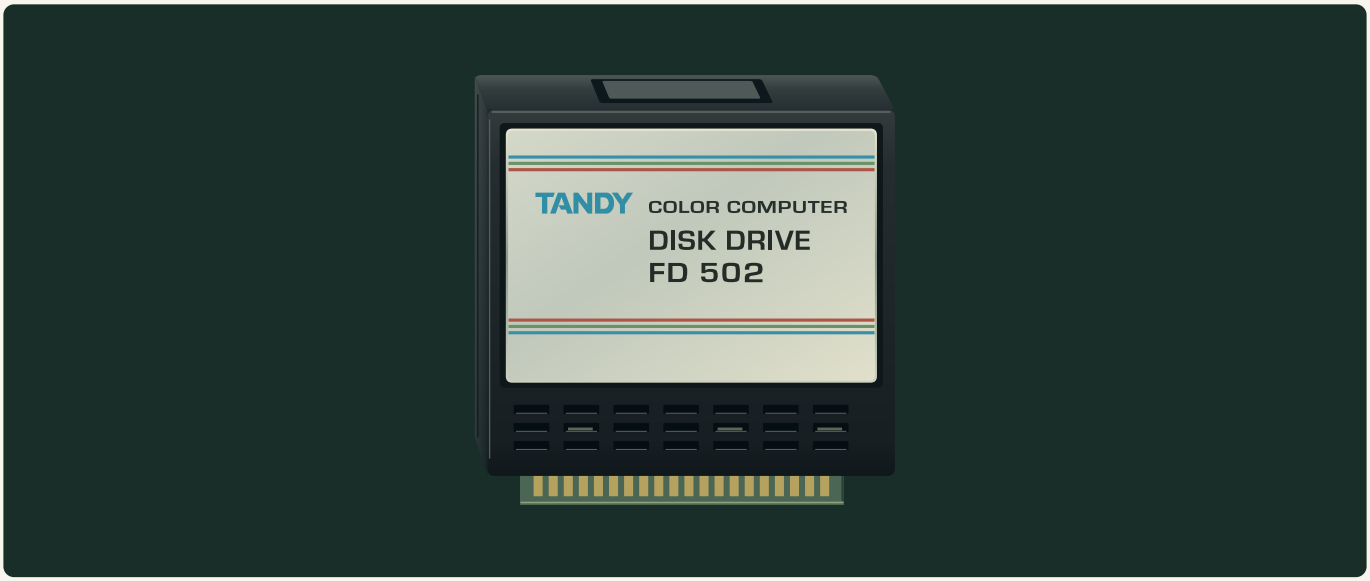
The CoCo 3 and CM-8 monitor presentation in VCC Plus's skeuomorphic System view was inspired by SockMaster's "pic of CoCo 3 and CM-8 monitor," published on CoCopedia. VCC Plus's artwork was designed and rendered independently. The remaining hardware illustrations were independently designed using historical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

Read this book online

CMOC license notice · LWTools COPYING notice

Programming in C on the CoCo 3 — CMOC and Its APIs

Give Mars a key. Count meals for two planets. Then put a program and its supplies on a floppy someone else can run. Along the way, stop the processor and see exactly what your C code handed it.



Black Tandy FD-502 disk controller cartridge with a silver face label and exposed card edge.

Brian K. Holman

Book 6 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Give Mars a key 3
Give the CoCo its colors 6
Count cargo for two worlds 9
Hand a function one shipment 12
Build a disk from a folder 16
Follow a word through FD0 19
Find the file that went missing 38
Choose a library for a job 41
Hand someone a disk they can run 57
Find a call 60
CMOC at the desk 191
Source shelf and further reading 195
Publication and artwork notices 198

Give Mars a key

Chapter 1 of 9

Somebody has to write the part of a game that waits at the title screen. Our whole first game is that part. Give the CoCo a **1** and it greets Earth. Give it a **2** and it greets Mars. By the time it answers, one byte will have crossed from your keyboard into a decision you can find in the source.

Make the first run

1. Begin with the **Disk BASIC** system profile, running at its **OK** prompt. Save any program you care about before continuing. Direct Run uses the CoCo's program memory.
2. On the Mac choose **Develop → C Development…**. Click **New**, choose **Hello World**, give the new folder a name such as **TwoWorlds**, and select **Create Project**. The editor opens **main.c**. The bundled starter and the copy in your folder have the same starting text. You can also save the complete planet source from this book.
3. Leave the configuration on **Debug** and click **Build**. Look for **Build succeeded**. You now have two things you can keep on the Mac. The source file is for a person to read and change. The resulting **main.bin** contains instructions for the CoCo processor. The CoCo is still at its prompt.
4. Click **Run** at the top of the C window. Confirm **Run** when the app explains that it will replace program memory. When the question appears on the CoCo, click inside its display and press **1**. A single key is enough. Look for **Hello Nice Blue Dot World!** and then **OK**.
5. Return to the **C Development** window. Click **Run** again, confirm and choose **2** at the CoCo. You used the same compiled program twice. Which part did the key change?

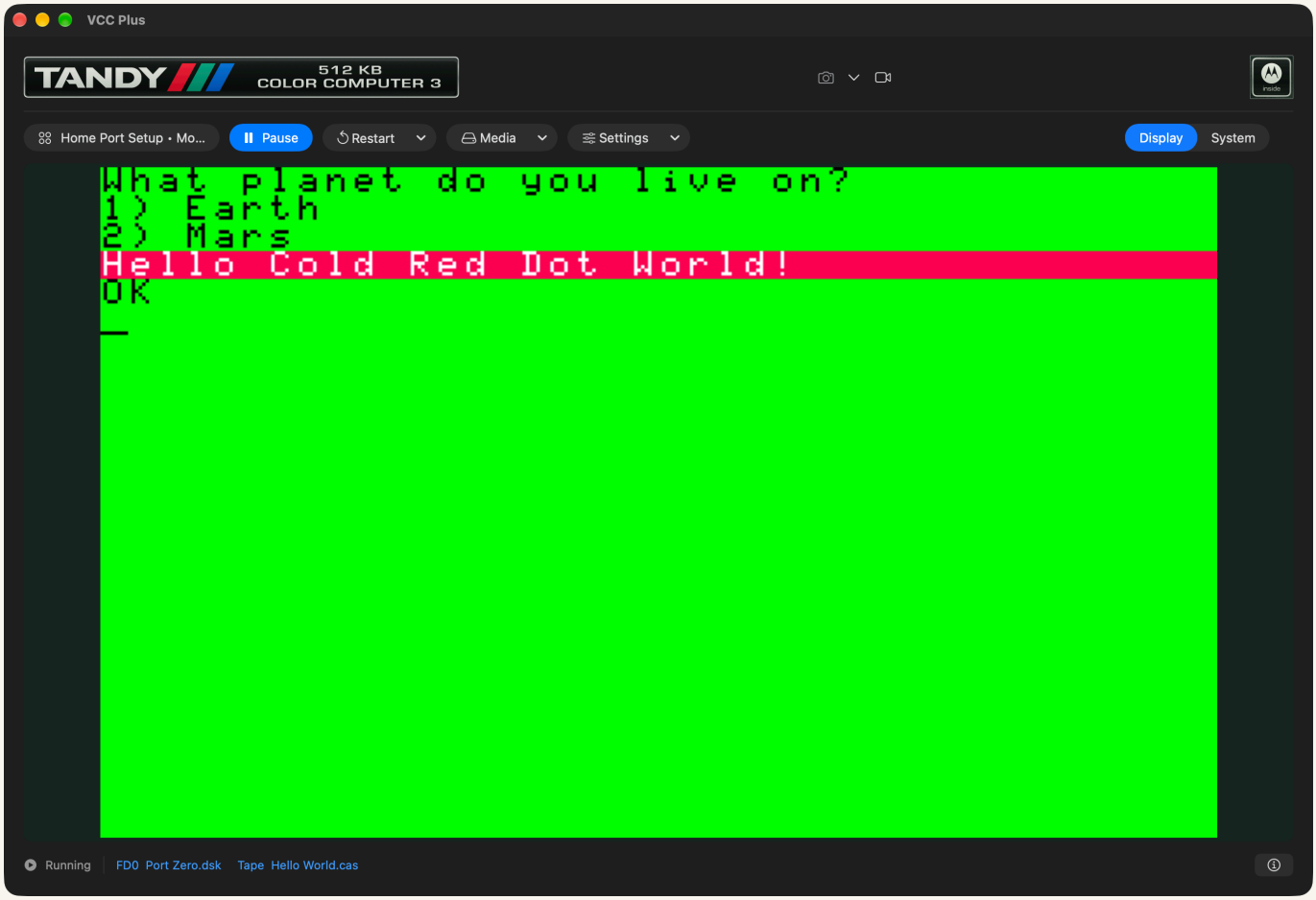


Figure 1: The running VCC Plus display after pressing 2 shows the menu, Hello Cold Red Dot World! on a red line, and BASIC's OK underneath.

If the guest sits at the question, first check where the keyboard is going. Choose **Input → Focus CoCo Display** and press **1** or **2**. If you see a flashing caret in **main.c**, you are typing into the Mac editor. A freshly created starter has no source breakpoints. You can leave the **Run to Breakpoint** button at the bottom alone for now.

Read the keystroke in C

```
int main(void)
{
    byte choice;

    width(40);
    printf("What planet do you live on?\n1) Earth\n2) Mars\n");
    do
    {
        choice = inkey();  /* No Enter needed. Ignore other keys. */
    } while (choice != '1' && choice != '2');
```

`main` is the function where the program starts its own work. Its braces enclose that work. The parentheses in `main(void)` tell CMOC the function takes no arguments. The local variable `choice` holds one byte. The call to `width(40)` asks the CoCo 3 to switch to a 40-column text display. The double quotes around the question enclose a string, and each `\n` in that string moves printing to a new line.

Start at **do** and read downward as the processor would. The call to `inkey()` checks the keyboard. With no key pressed it returns zero. Otherwise it supplies a character code. The single = stores that result in `choice`. At **while**, `!=` means "different from," and `&&` means both tests must hold. If the value differs from both `'1'` and `'2'`, execution goes around again. Imagine pressing M. Both comparisons are true, so the loop checks the keyboard again.

The single quotes matter. `'1'` is the code for the character printed on the key. The number `1` is an integer with a different value. Comparing the returned character to another character lets the program recognize exactly what was typed.

Who chooses the greeting?

Find `if (choice == '1')` in your copy of **main.c**. The double equals sign tests whether two values match. Trace the first branch as far as its closing brace. Now put your finger on **else** and follow the other branch. The earlier loop guaranteed that reaching this second branch means the player chose Mars. Both paths meet again at the final text-attribute call, then `return 0;` ends the function and control returns to BASIC.

The two lines at the very top beginning with `#include` bring in declarations that CMOC uses while compiling. The header **cmoc.h** declares `printf`, which produced the question and greeting. The header **coco.h** declares `width` and `inkey`, and defines the name `byte`. These names give you a way to look up what a call expects before you change it. You'll try that on the colors next. If you've already written the same decision in BASIC, trace the two choices in Planet Explorer and compare the moment each program accepts its answer. To follow the key all the way down to the processor, catch the shared greeting in assembly.

Give the CoCo its colors

Chapter 2 of 9

Why does just the greeting change color while the next line returns to green? One call chooses how to draw the next letters. A second call chooses the color assigned to a palette slot. Earth and Mars share a slot, even though their colors differ.

Find the two numbers

```
    attr(3, 6, FALSE, FALSE);  /* White text on background slot 6. */
    if (choice == '1')
    {
        palette(6, 10);  /* Blue */
        printf("Hello Nice Blue Dot World!\n");
    }
    else
    {
        palette(6, 37);  /* Red */
        printf("Hello Cold Red Dot World!\n");
    }
    attr(0, 0, FALSE, FALSE);  /* Back to BASIC's usual colors. */
```

The starter is already in 40-column mode when it reaches this code. In `attr(3, 6, FALSE, FALSE)`, the first two arguments select the foreground and background for subsequent printing. Here the foreground gives light lettering. Six chooses a background slot. The two **FALSE** arguments leave blink and underline off.

Now follow slot **6** through the branches. Choosing Earth stores color value **10** in that slot. Choosing Mars stores **37** there. Whichever branch you take, the letters are printed with the same text attributes, and their background uses the value currently in slot 6. The CoCo 3 has sixteen palette slots, numbered 0–15. Each slot can hold one of 64 color codes, numbered 0–63. The text mode allows a smaller selection of foreground and background indices for each character.

What you change	What to watch
The background argument of **attr**	Which slot subsequent lettering asks to use
The color argument of **palette**	The visible color wherever that slot is used
The final **attr** call	The attributes BASIC inherits for printing its prompt

Notice the last line of the listing. The starter returns the printing attributes to their usual settings. It leaves the chosen value in palette slot 6. Earlier lettering using that slot can still show the color when the **OK** prompt appears below it.

Compare two monitor signals

Select **Display → RGB Monitor** in VCC Plus. Run **TwoWorlds** and press **1**. Then choose **Display → Composite TV**, run the same build and press **1** again. You can compare the two real runs below. Keep an eye on the whole screen, including its usual green. Both runs reached the same branch and printed the same letters. The monitor's interpretation of the video signal changed the shades you saw.

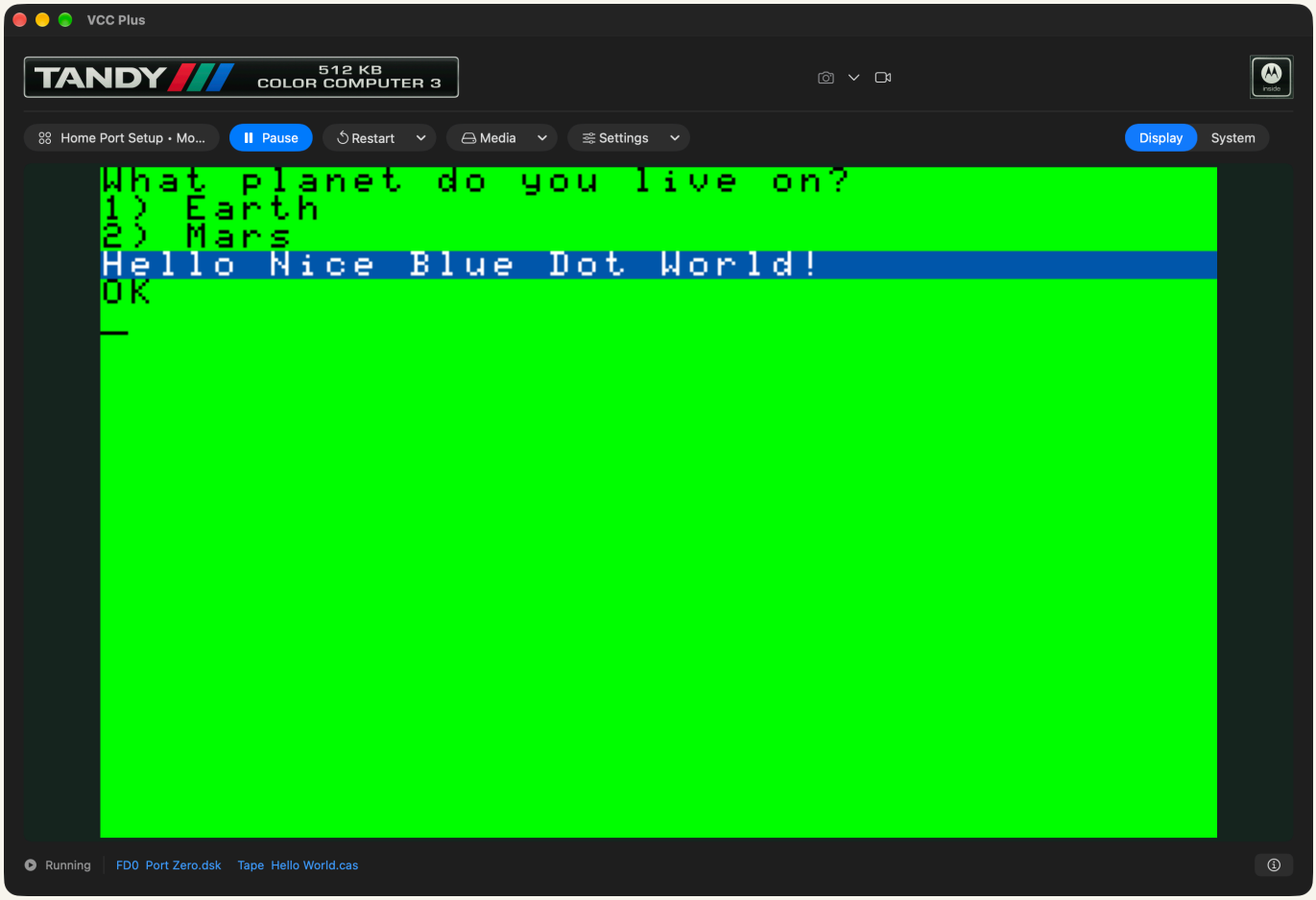


Figure 2: Earth under RGB Monitor. The greeting uses a blue band and the BASIC background is bright green.

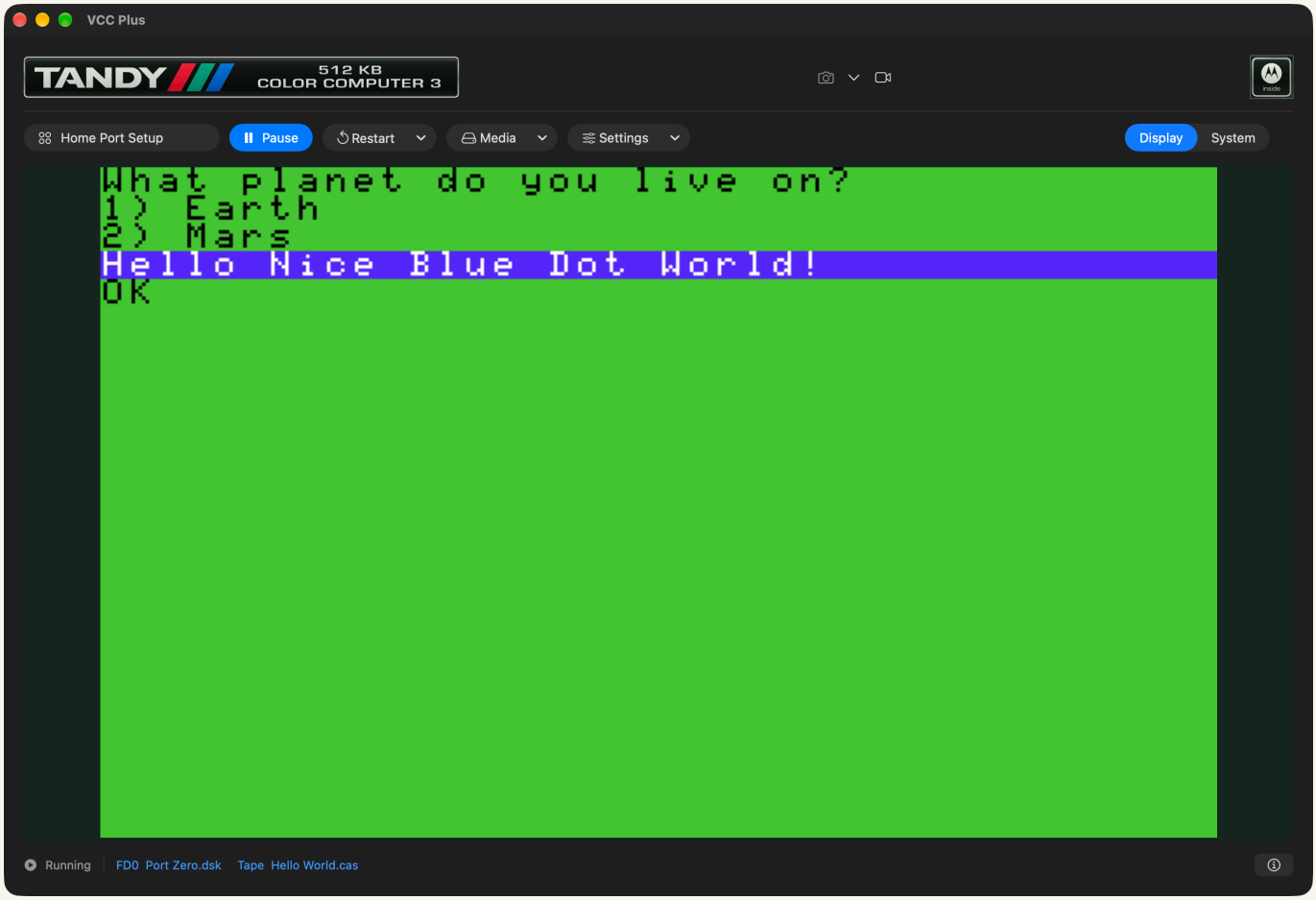


Figure 3: Earth under Composite TV with the same build, key and palette value. The band is violet-blue and the green is softer.

Give Mars Earth's blue

1. Make a separate **Hello World** project for this experiment. In **main.c**, place the insertion point inside the argument list of `palette(6, 37)`. To see what the second argument means, request **Show Parameter Hints** beside the editor or press **Control-Shift-Space**. The declaration identifies **slot** and **color**. You can also put the caret in the name **palette** and use **Control-Command-J** to open its declaration in the read-only **coco.h** header. Use **Back** to return.
2. In the Mars branch change only **37** to **10**. Leave the greeting and the slot alone. Save, build, run with **RGB Monitor**, and press **2**. You should read **Hello Cold Red Dot World!** against the blue used for Earth. Its words and color come from different instructions.
3. Change **10** back to **37** in that branch, save and rebuild. Choose the monitor signal you normally use.

A compiler can accept both versions. Which version tells the story you meant to tell? That question belongs to the person running the program. In the next experiment the words will stay the same while one changed number alters the total.

Count cargo for two worlds

Chapter 3 of 9

Earth ordered twelve crates with twenty meals in each. Mars ordered eight crates with fifteen meals in each. The ship has one manifest and two stops to make. How many meals should the cook set aside before departure?

Load a copy of the manifest

Choose **New → Hello World** in **C Development** and create a separate project named **CargoCount**. Save the Cargo Count source in your practice folder. Open that saved source in a text editor and copy its contents. In the project's **main.c** editor select all the starter text and replace it with the Cargo Count source. Keep the name **main.c** because the project's Makefile builds that name. Choose **Save All**, leave **Debug** selected, then **Build** and **Run**. This program needs no floppy and asks for no keys.

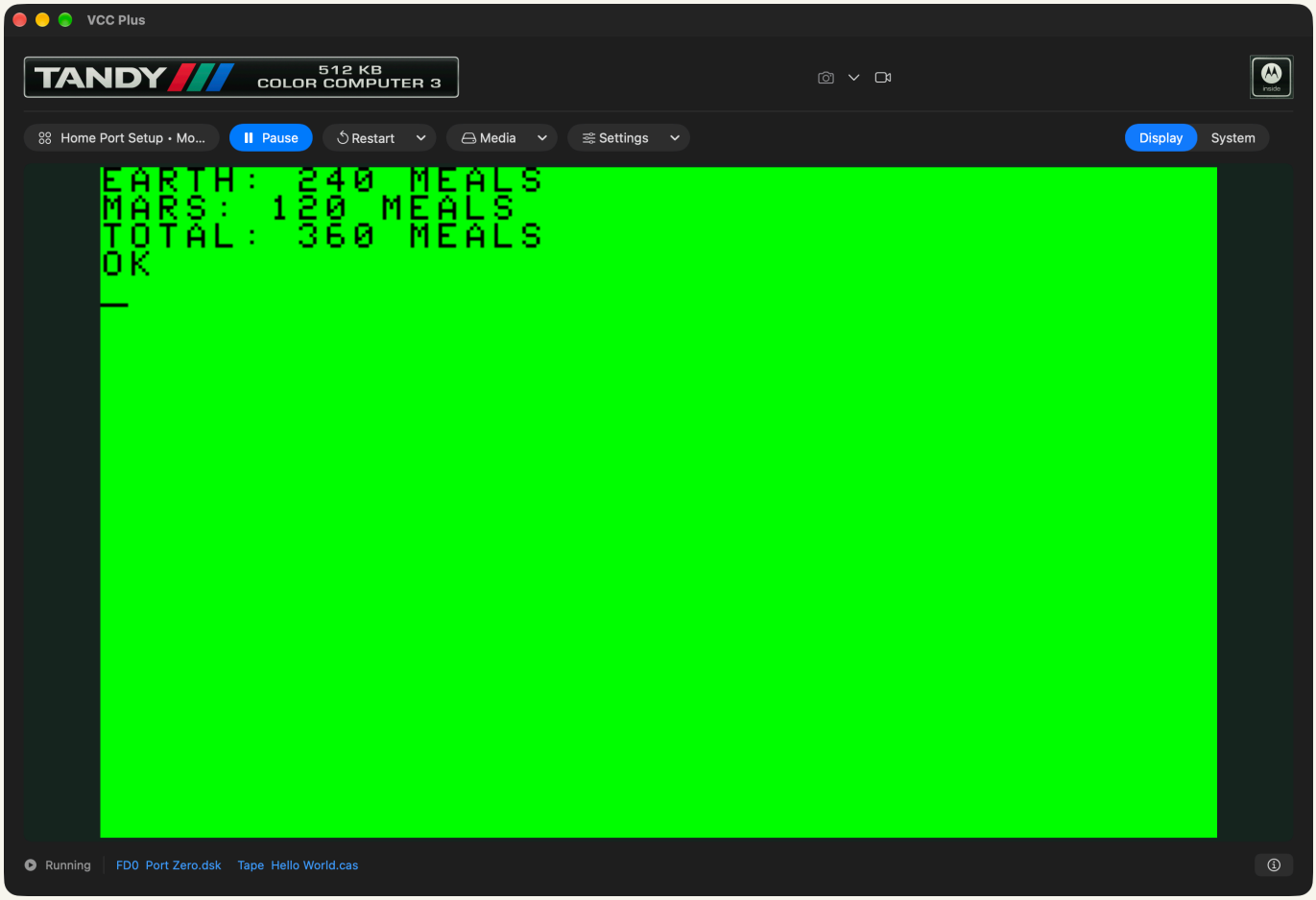


Figure 4: The real guest has printed EARTH: 240 MEALS, MARS: 120 MEALS and TOTAL: 360 MEALS, then returned to OK.

Read one row as a shipment

```
struct Shipment
{
    const char *destination;
    byte crates;
    byte mealsPerCrate;
```

```
};

static const struct Shipment manifest[] =
{
    { "EARTH", 12, 20 },
    { "MARS", 8, 15 }
};
```

A **structure** groups different pieces of information about one thing. A **Shipment** has a destination, a count of crates and a count of meals per crate, in that order. Each initializer provides those three values. The double quotes make each destination a string ending with a zero byte. The `const char *` member holds its address so the program can find the letters to print.

The square brackets in **manifest[]** make an **array** of shipments. CMOC counts the two initializer rows to determine its length. Index zero is Earth and index one is Mars. A **byte** in this program is an unsigned 8-bit value, large enough for 0 through 255. Each crate count fits. An unsigned **word** has 16 bits and can hold 0 through 65,535. That's the type chosen for a shipment's meal count and the ship's running total. Even though **240** and **120** fit in single bytes, their sum **360** exceeds 255.

Work the loop by hand

```
int main(void)
{
    word total = 0;
    byte row;

    width(40);
    for (row = 0; row < sizeof(manifest) / sizeof(manifest[0]); ++row)
    {
        word meals = mealsFor(&manifest[row]);
        printf("%s: %u MEALS\n", manifest[row].destination, meals);
        total += meals;
    }
    printf("TOTAL: %u MEALS\n", total);
    return 0;
```

Read the three parts inside **for** in order. Start with `row = 0`. Check that the index is less than the number of shipments. After processing a row, `++row` advances to the next one and the check happens again. The expression `sizeof(manifest) / sizeof(manifest[0])` divides the storage of the entire array by the storage of one element. It works here because the array itself is in scope. Using the size of a pointer in its place would answer a different question.

For the first pass, `mealsFor` gives back **240**. The two placeholders in `printf` use the two arguments after the format string. **%s** follows the destination string to its zero terminator. **%u** prints the meal count as an unsigned decimal number. The shortcut `total += meals` adds this shipment to the running count. Fill in the second row before looking at the screen.

Row	Destination	Crates × meals per crate	Total after this row
0	EARTH	12 × 20 = 240	240
1	MARS	8 × 15 = 120	360

Send one more crate

Change **8** to **9** in the Mars initializer, and nothing else. Predict the two destination counts and the ship's total before you choose **Save All → Build → Run**. Earth still needs **240**. Mars now needs **135**, so the final count is **375**. The addition to the grand total was exactly one crate's worth. If your screen still says **360**, check which copy of **main.c** you edited and that the build status says **current**.

Set Mars back to **8**, save, and build in **Debug** again. Keep this project open. You are about to look at one delivery before the program prints it.

Hand a function one shipment

Chapter 4 of 9

The cook handed the pilot a single number, **360**. How could you tell which shipment the program was counting *before* a number appeared on screen? Stop the running program while its helper is working, then look back at the row that called it.

Stop before the first count

Use the **CargoCount** project from the previous chapter with the original two rows. If you are starting here, get the complete cargo source and follow the project setup. Select **Debug**, save and build. The debugger needs the source and compiled binary from the same build.

1. At the bottom of **C Development**, open **Source breakpoint**. Pick **Line 33**, the assignment that calls `mealsFor(&manifest[row])` in the supplied file. Use the statement to find it if you've added your own comments.
2. Click **Run to Breakpoint** and confirm. That command targets the line you selected. Clicking an ordinary **Run** with no added breakpoints lets the program finish immediately.
3. At the pause choose **Into**. You may first stop on the function header at line 20. Press **Over** once if necessary to reach its multiplication at line 22. A stop on the function header can be too early for CMOC to describe its parameters. Wait until the highlighted source and pause message both name the line you want to inspect.

```
/* A pointer lets us read one row without copying the whole structure. */
static word mealsFor(const struct Shipment *shipment)
{
    return (word) shipment->crates * shipment->mealsPerCrate;
}
```

The ampersand at the call site means “the address of.” On the first pass `&manifest[row]` gives the address of the Earth shipment. The star in `const struct Shipment *shipment` declares a pointer that the function can use to read that shipment. The arrow in `shipment->crates` follows the pointer and selects its crates member. The cast `(word)` makes the first factor explicitly an unsigned 16-bit integer for this calculation.

Open the pointed-to shipment

Select **Variables** in the lower pane. In **Inspect frame**, choose **0: mealsFor** if necessary. Expand **shipment**. Expand the **Pointee** underneath it to inspect the structure at that address. Look for **crates = 12** and **mealsPerCrate = 20**. The hexadecimal address beside a pointer may differ between builds. Its members tell you which delivery it describes.

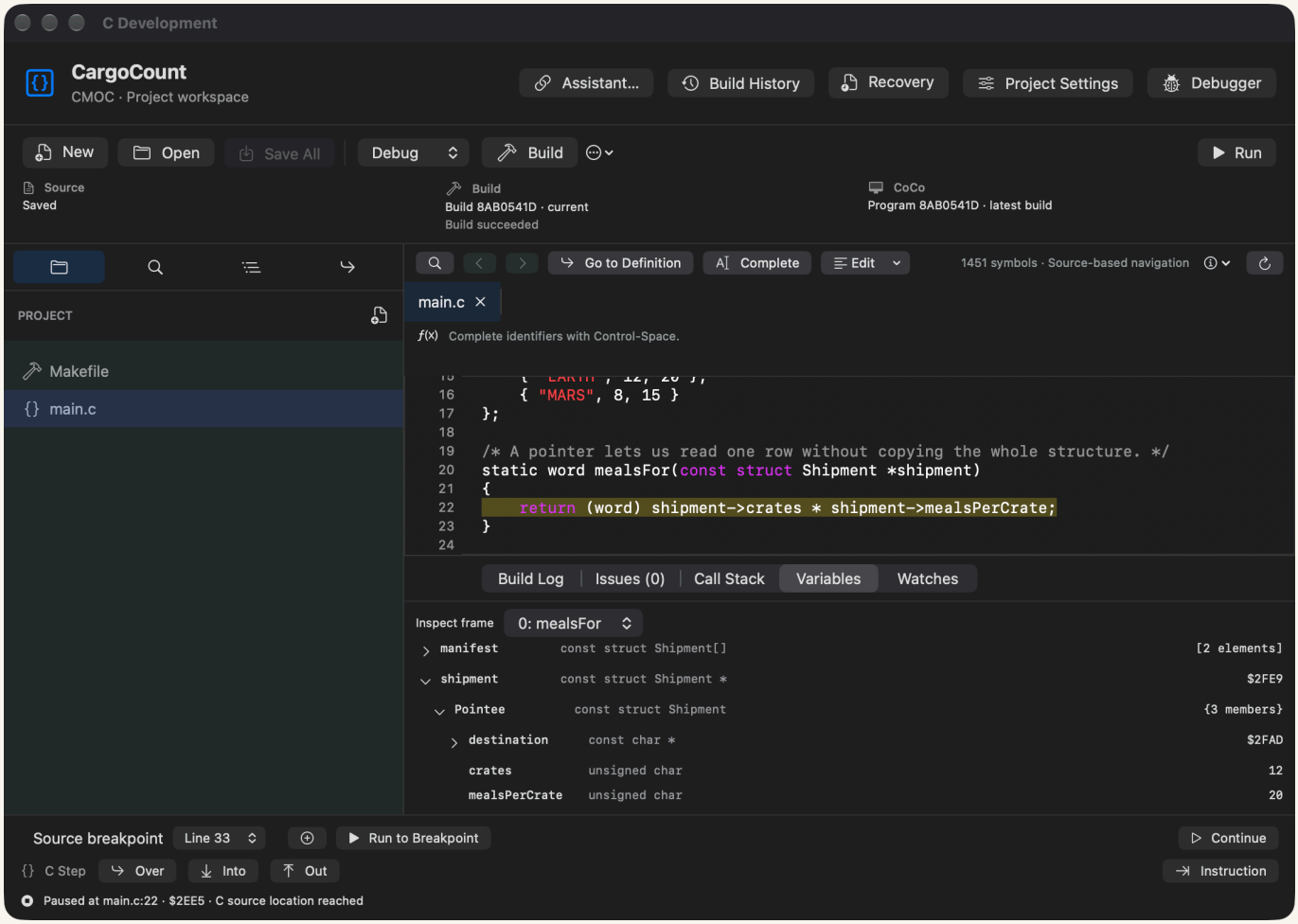


Figure 5: In the paused C Development window, line 22 multiplies the two members. Variables shows shipment expanded through Pointee to crates 12 and mealsPerCrate 20.

To keep one value in sight, select **Watches**, enter `shipment->crates` in the expression field and press Return. In this frame on the first delivery it should read **12**. Watching an expression reads paused memory. You can inspect a member safely without running a second copy of the function. A watch written with a parameter name belongs to the frame where that name has meaning.

Look back at the pilot's row

Select **Call Stack**. Frame **0** is the function currently executing. Frame **1** is **main** at the call that brought you here. Select frame **1**, then **Variables**. The source moves back to the assignment in **main**. You should now find **row = 0** and **total = 0**. Execution still waits inside **mealsFor**.

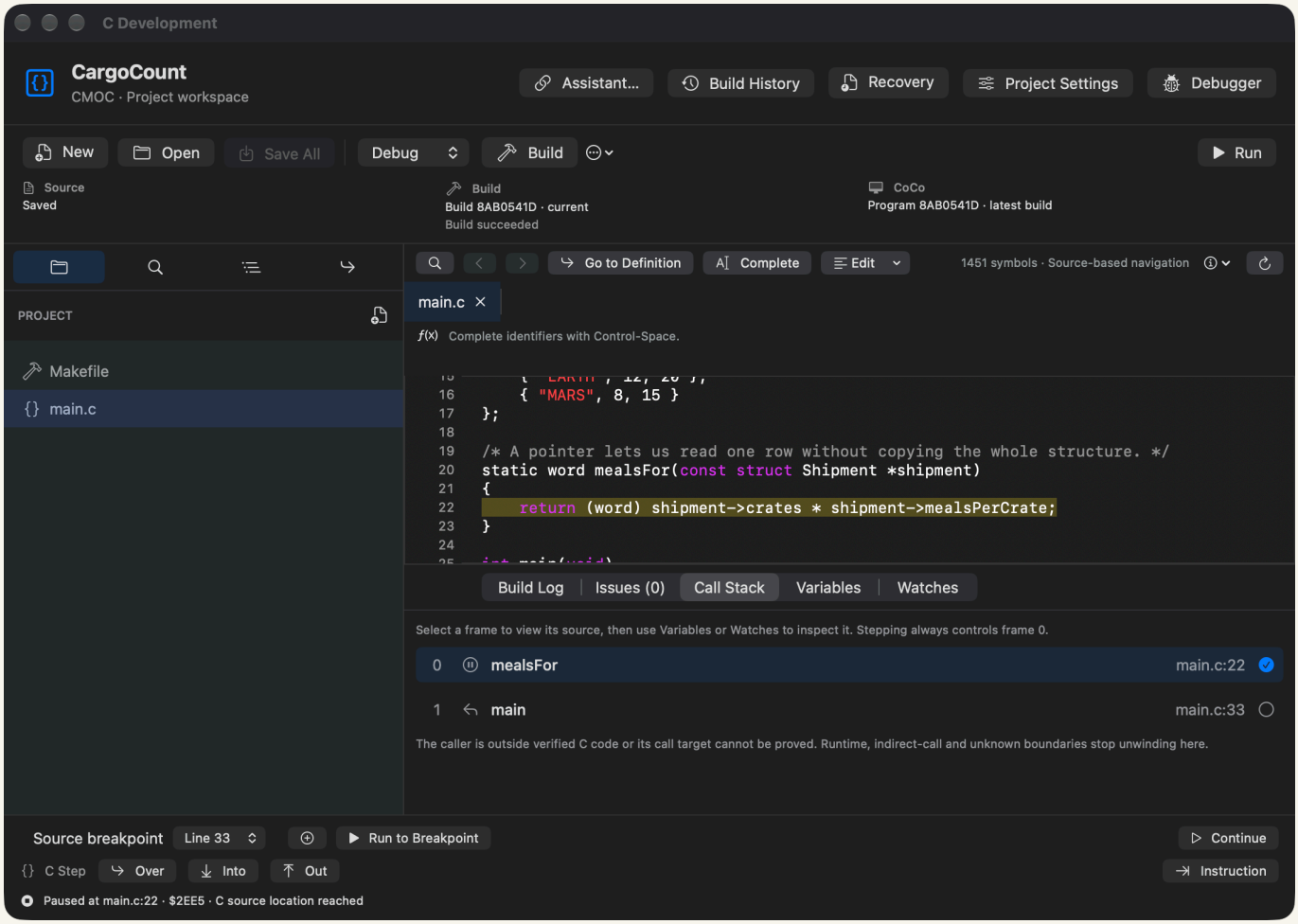


Figure 6: Call Stack shows mealsFor stopped at line 22 and its caller main at line 33.

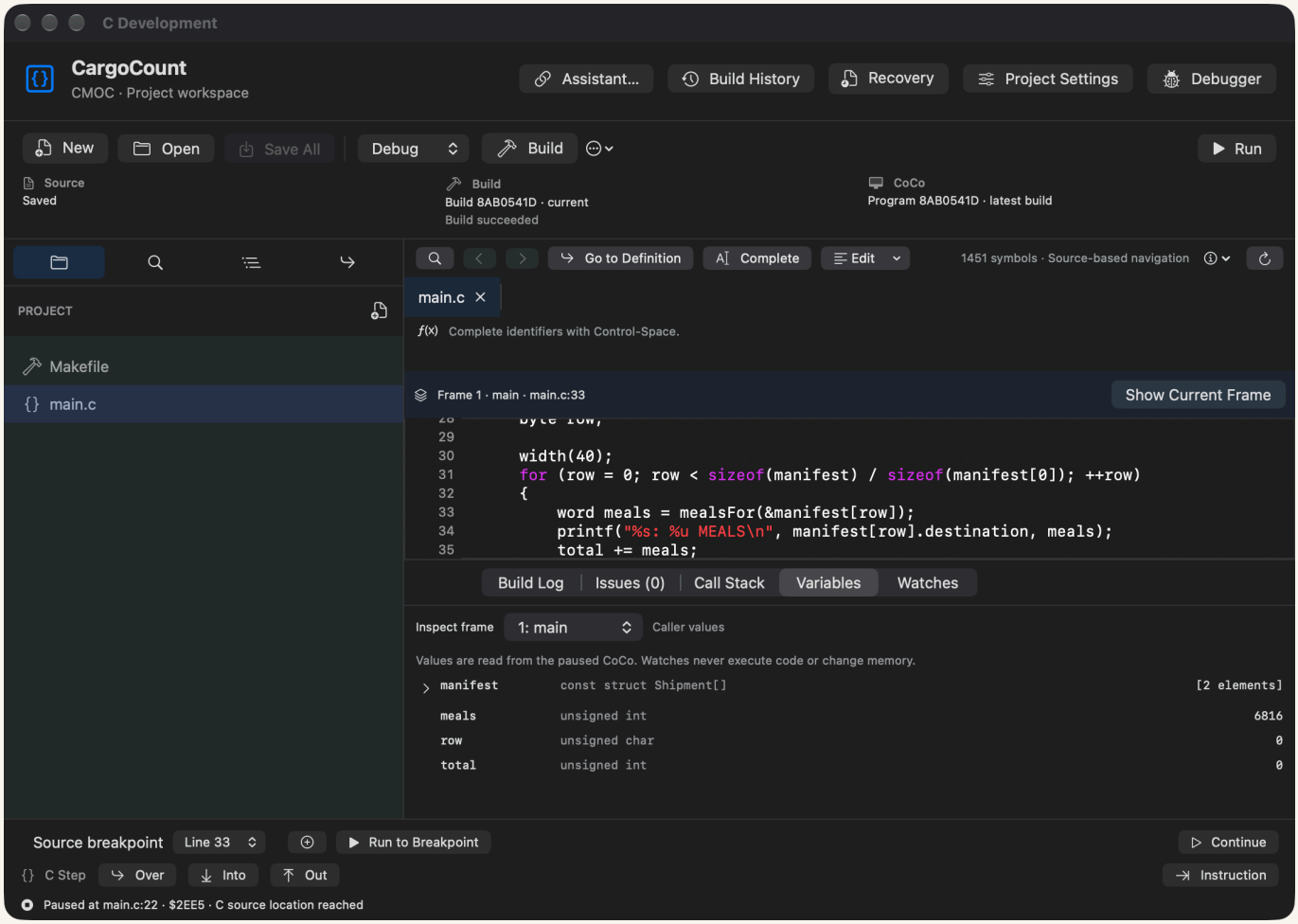


Figure 7: Selecting frame 1 shows main at its call site. The caller's row and total are both zero. Its local meals has not yet received the answer.

There may already be a number beside the caller's **meals**. The right-hand side of the assignment is still running. Until the helper returns, that storage can hold a leftover value. Select frame **0** again. **Out** finishes the helper and returns to its caller. If the assignment is still highlighted, step **Over** to the following line before reading **meals**. Its first assigned value is **240**. Continue to see the Earth line, or inspect the next visit at the call site. During the Mars call the caller's **row** is **1** and its accumulated **total** is **240**. Finish with **Continue**, resuming again if the same source stop is encountered.

How long does an address last?

The manifest has **static** storage. Its two rows remain in memory throughout the program, so a helper can read the selected row whenever it is called. **const** on the pointed-to shipment prevents the helper from modifying that row through this pointer. The local **row** and **total** belong to this invocation of **main**. The local **meals** belongs to the block inside the loop. When the block ends, another use of that storage is free to overwrite it.

C lets you form a pointer without attaching an element count. The loop protects the access by checking the row *before* handing off the address. Valid indices for this manifest are **0** and **1**. Reading `manifest[2]` would step outside the array, even if the debugger could show bytes at that address. Keep the bounds check and the lifetime of the object in view whenever a pointer crosses a function boundary.

You've now examined source on the Mac and data in the CoCo's RAM. In the next project, you'll follow the program into a data file on the floppy in FD0.

Build a disk from a folder

Chapter 5 of 9

Open a brand-new **Hello Disk** project. The guest will display five words. Only the headings live in the C files you first see. To find the five words, follow the build backward from the floppy to the Mac.

Meet the crew

Choose **Develop → C Development… → New → Hello Disk**. Give the project its own writable folder. In the editor's file list, open each of these files in turn.

File on the Mac	Its job	Where its contribution ends up
`main.c`	Starts CoCo support, asks for the heading and the words, prints the result.	Instructions in HELLO.BIN
`src/display.c`	Prints the two fixed headings.	Instructions and text in HELLO.BIN
`src/disk.c`	Opens the guest file and prints the text it reads.	Instructions and a reserved buffer in the running program
`include/hello.h`	Declares the two functions and the shared count, so each C file knows their types.	The compiler checks the callers against these declarations.
`assets/words.txt`	Holds the five editable words, one per line.	Supplies both the compiled count and the disk data
`tools/build-data.pl`	Reads that list on the Mac and produces two files.	Generated C source and a 256-byte WORDS.DAT

Here is the whole conductor's score, from the installed template.

```
/* SPDX-License-Identifier: GPL-3.0-or-later */
#include <cmoc.h>
#include <coco.h>
#include "hello.h"

int main(void)
{
    unsigned char status;
    initCoCoSupport();
    setHighSpeed(FALSE);
    showHeading();
    status = showDiskWords();
    printf("DISK RESULT: %u\n", status);
    printf("GENERATED WORD COUNT: %u\n", generatedWordCount);
    return status;
}
```

Read the last two print calls aloud. One number comes back from a function at run time. The other was counted when you built the program. They can tell different stories.

Follow the arrows back to their inputs

Open **Project Settings → Preview** or read the project's Makefile. Find the rule starting with `HELLO.dsk:`. The name on the left is a *target*, something Make knows how to produce. The items on the right must be ready first. The indented lines beneath it are the recipe. Here is the route to take through this particular file:

1. To make **HELLO.dsk**, the recipe creates a fresh image, installs a small DOS loader, and puts **HELLO.bin** and **generated/WORDS.DAT** in its directory.
2. To make **HELLO.bin**, the linker joins four compiled pieces and resolves the disk calls using **decbfile**. Three pieces correspond to the handwritten C files. The fourth comes from **generated/word-count.c**.
3. The grouped rule for **generated/word-count.c** and **generated/WORDS.DAT** runs once to create both from the same word list. Open **Resources** to see the inputs, outputs and Perl command. One output will become part of the program. The other travels beside it.
4. To change the cargo, edit **assets/words.txt**. An edit inside a generated file can disappear on the very next build.

A header is useful here for a different reason than a library. The declaration in hello.h tells the compiler how to call showDiskWords. The object built from disk.c supplies that function to the linker. Likewise, decbfile.h declares the disk calls and the library archive supplies their implementations.

Pick the button that performs your experiment

Start a **Disk BASIC** session with the FD-502 available. Save work in the guest before launching a different program.

Action	What happens with these projects
Build	Runs the configured build target on the Mac. Hello Disk specifies `disk`, so this builds its image as well as its binary. Hello World normally builds a binary.
Build Disk	Requests the configured disk target explicitly. In the supplied Hello Disk project it is also `disk`.
Run	With binary launch selected, the workbench loads the executable into the CoCo and enters it directly. Earth or Mars needs only this.
Launch Disk	With disk launch selected, the workbench uses the built image, asks about replacing FD0 as needed, and sends the project's BASIC launch command. Hello Disk uses `DOS`.

Choose **Build Disk**. Read the last successful target in the build output. Choose it a second time without editing anything. Make can report that the target is already up to date. **Rebuild** is useful when you specifically want fresh compilation. **Show Build Products** takes you to the result on the Mac. The app builds in its own work area, so looking for a newly written HELLO.dsk among your handwritten sources can send you on a wild-goose chase.

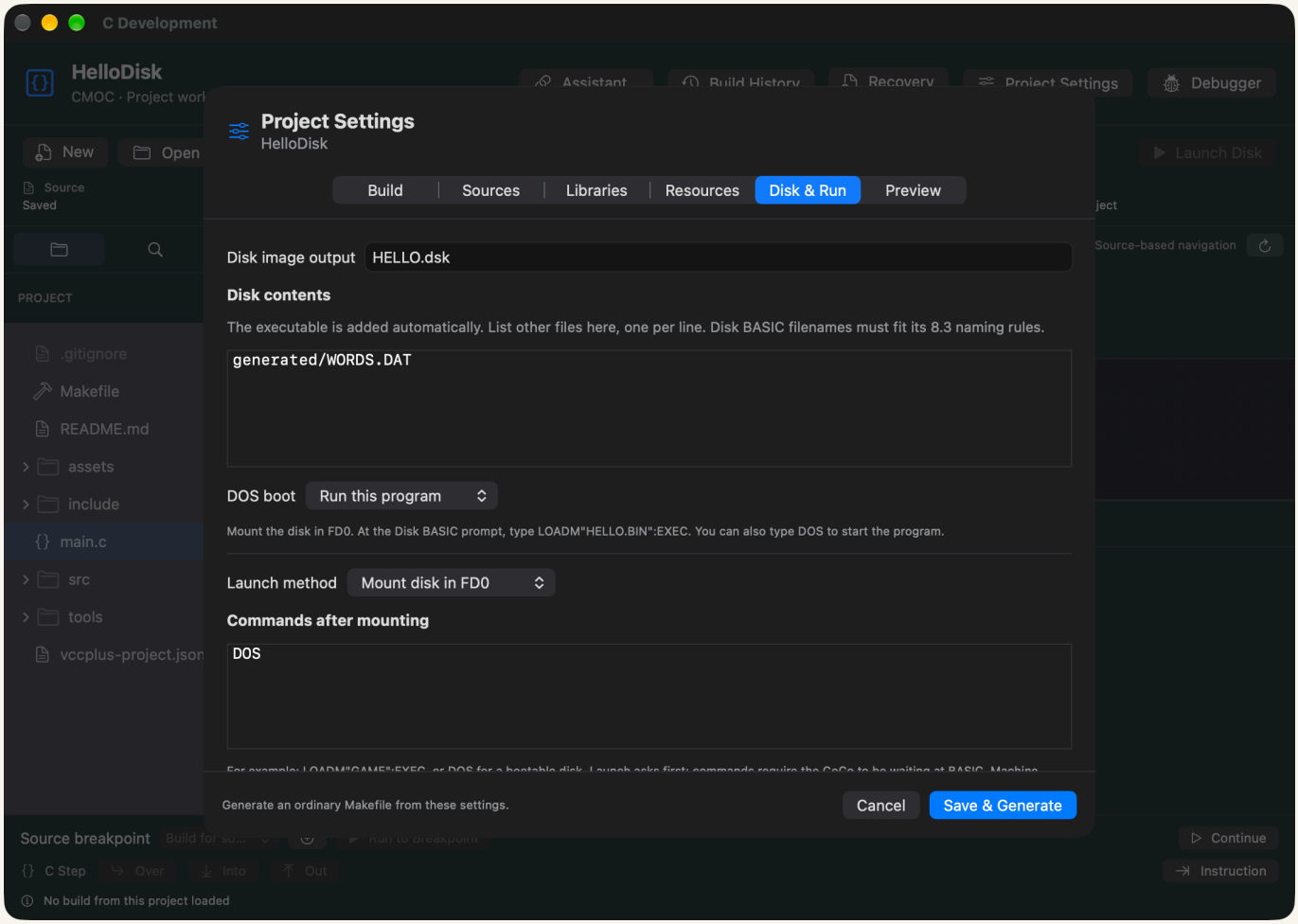


Figure 8: In the C workspace, Disk & Run lists generated/WORDS.DAT as an extra file. The executable is included automatically. The disk will occupy FD0 and DOS will launch it.

Next choose **Launch Disk**. Attend to any floppy-replacement prompt. On the guest you should see five words and two numbers. Keep the disk and project open for a trip through the sector reader.

Follow a word through FDo

Chapter 6 of 9

When **WORDS READ FROM THE FLOPPY:** appears, the CoCo has printed a string built into the executable. The next line, **HELLO**, has made a longer journey. It began as a line of Mac text, crossed into a file on a virtual floppy, and came back as a sequence of bytes in guest RAM.

Set a desk for the disk library

Open src/disk.c in your Hello Disk project. Its three file-scope declarations reserve a drive record, a file record and a 256-byte reading area. Each uses `static`. The objects live for the whole program and belong to this source module. Reading a sector into a large local array on every call would spend part of the CoCo's limited stack instead.

```
static DECBDrive drives[1];
static DECBFile file;
static byte words[256];

byte showDiskWords(void)
{
    byte status = decb_init(drives, 1);
    if (status != DECB_OK)
        return status;
    status = decb_registerDrive(0);
    if (status == DECB_OK)
        status = decb_openSectorFile(&file, 0, "WORDS.DAT");
```

The 1 in decb_init tells the library how many drives it has room to manage. The 0 in decb_registerDrive identifies the guest drive. One record is enough for drive **zero**. The file-open call then asks that drive's directory for WORDS.DAT and fills in the caller's file record. On a real CoCo the controller would reach a physical floppy. Here the same guest routine reaches the disk image assigned to FD0. See where a file lives inside a floppy image.

Read a disk sector

```
    if (status == DECB_OK)
    {
        status = decb_readSector(&file, words, 0);
        words[255] = 0;
        if (status == DECB_OK)
            printf("%s\n", words);
        decb_closeSectorFile(&file);
    }
    decb_shutdown();
    return status;
}
```

The last argument to decb_readSector selects logical sector **0 of the open file**. The words array supplies the memory into which the library can write all **256 bytes**. The CoCo may store that sector anywhere allocated to the file. You didn't have to calculate a track or look through a disk-image header.

Once a successful read fills the buffer, printf walks bytes from the start until a zero byte ends the string. Each word in this particular file is followed by a carriage return. The generator writes a final zero and pads the sector; the reader also places a zero in the last byte. A different binary file could contain zeros in its data, so this string-

printing technique belongs to this text format. For a longer file the program would have to read more than one logical sector and interpret the bytes it receives.

Trace the cleanup with your finger. If open succeeds, the program closes the file even when a read fails. If open fails, it skips that close. After successful initialization it shuts the disk library down before leaving. The sample returns the status retained from registration, opening or reading. It doesn't report an independent close result.

Tell the two outputs apart

After **Launch Disk**, read these five guest lines in order: **HELLO**, **WORLD**, **CMOC**, **VCC**, **PLUS**. Beneath them you should see **DISK RESULT: 0** and **GENERATED WORD COUNT: 5**. You can find either number in the program without scrolling through the word list. One is a value returned by the disk operation; the other is the constant compiled into the executable.

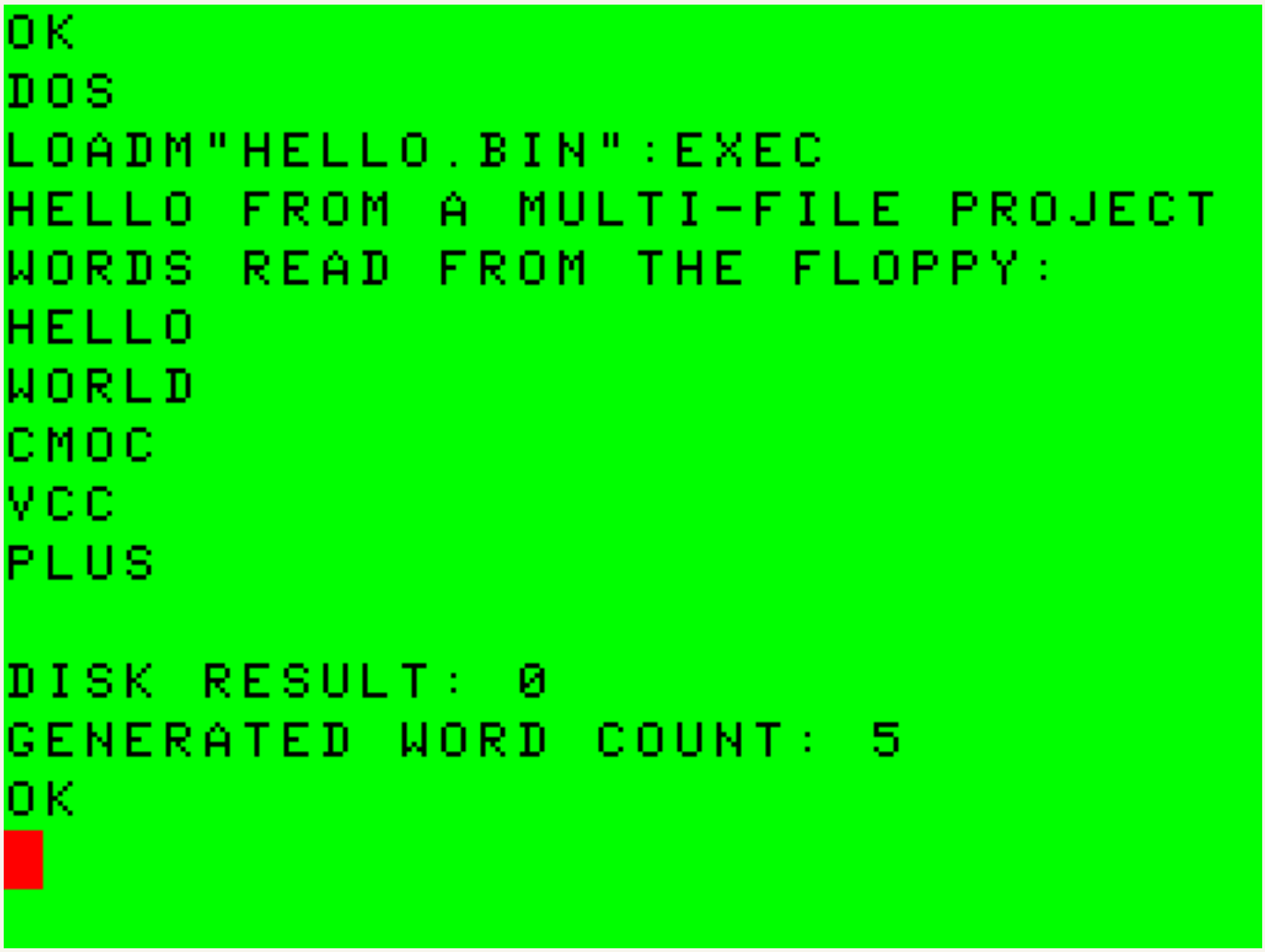


Figure 9: The running CoCo printed five words from FD0. The last two lines report the successful disk operation and the count compiled into HELLO.BIN.

When the guest is back at **OK**, type **DIR**. Find the entries for **HELLO BIN** and **WORDS DAT**. The directory prints the name and extension in separate fields. When the C code opens the file, it supplies the dot. If you give a friend only the binary, the heading and count go along for the ride. To take the five words too, give them the image.

Try predicting the two numbers when the program asks for a nonexistent file. Then test your prediction in the debugger.

Read a planet name across two sectors

Save the Planet Survey project, expand the ZIP, then open its **PlanetSurvey** folder in **Develop → C Development...**. Use the **Disk BASIC** system profile for this exercise. Choose **Build** and **Launch Disk**. The project mounts **SURVEY.dsk** in FD0 and sends `LOADM"SURVEY":EXEC`. Click the CoCo display and press **2**. Look for **SURVEY: MARS**, the traits **RED** and **COLD**, and orbit number **4**. Run the same command again and press **1** to read the Earth record.

Where do the sectors meet? In the original file, the letters **M** and **A** occupy bytes 254 and 255, counting from zero. **R** and **S** begin the next sector. Hello Disk fetched one sector into its own buffer. Planet Survey opens the same kind of disk file with fopen and reads one line at a time with fgets. The stream reads the next sector as the name goes by. Text mode also presents each guest carriage return as a C newline. That is the character the program removes before it formats the answer.

The loop stops when it finds your record. If **fgets** returns a null pointer first, check ferror while the stream is still open to distinguish a disk error from the ordinary end of the file. fclose releases the handle. The complete example uses **vcccompat** followed by **decbfile** under **Project Settings → Libraries**.

The filename **PLANETS.TXT** means drive 0 *inside the guest*. In Hard Drive BASIC, use DRIVE OFF to reach FD0. In CoCo SDC, copy SURVEY.dsk into your card folder and mount it as drive 0, then enter the same LOADM and EXEC command. Keep the image mounted until the program finishes reading it.

Read the list inside a planet record

Find the last line in **assets/planets.txt**.

```
MARS|RED,COLD|4
```

Imagine drawing a box around each field. The vertical bars divide the record into three boxes. Inside the middle box, the comma separates two traits. If you used both punctuation marks at once to split the whole line, you'd lose the distinction between a trait and the next field. Give every field a value. The tokenizer skips adjacent separators, so two bars together cannot stand for a blank field.

In **main.c**, follow the calls to strtok_r. The function stores a position in a pointer variable you supply. In **showRecord**, the pointer named **fields** holds your place in the record. The pointer named **traits** holds your place inside the list. Follow the five calls that return text on the Mars line.

Scan	Separator	What the call returns
Start the record	Vertical bar	MARS
Continue the record	Vertical bar	RED,COLD
Start the traits inside that field	Comma	RED
Continue the traits	Comma	COLD
Resume the record	Vertical bar	4

Which saved position gets you back to **4**? After a third request for a trait returns **NULL**, `strtok_r(NULL, "|", &fields)` picks up the outer record where you left it. The **4** is still text at this point. strtoul converts it to the number the program prints.

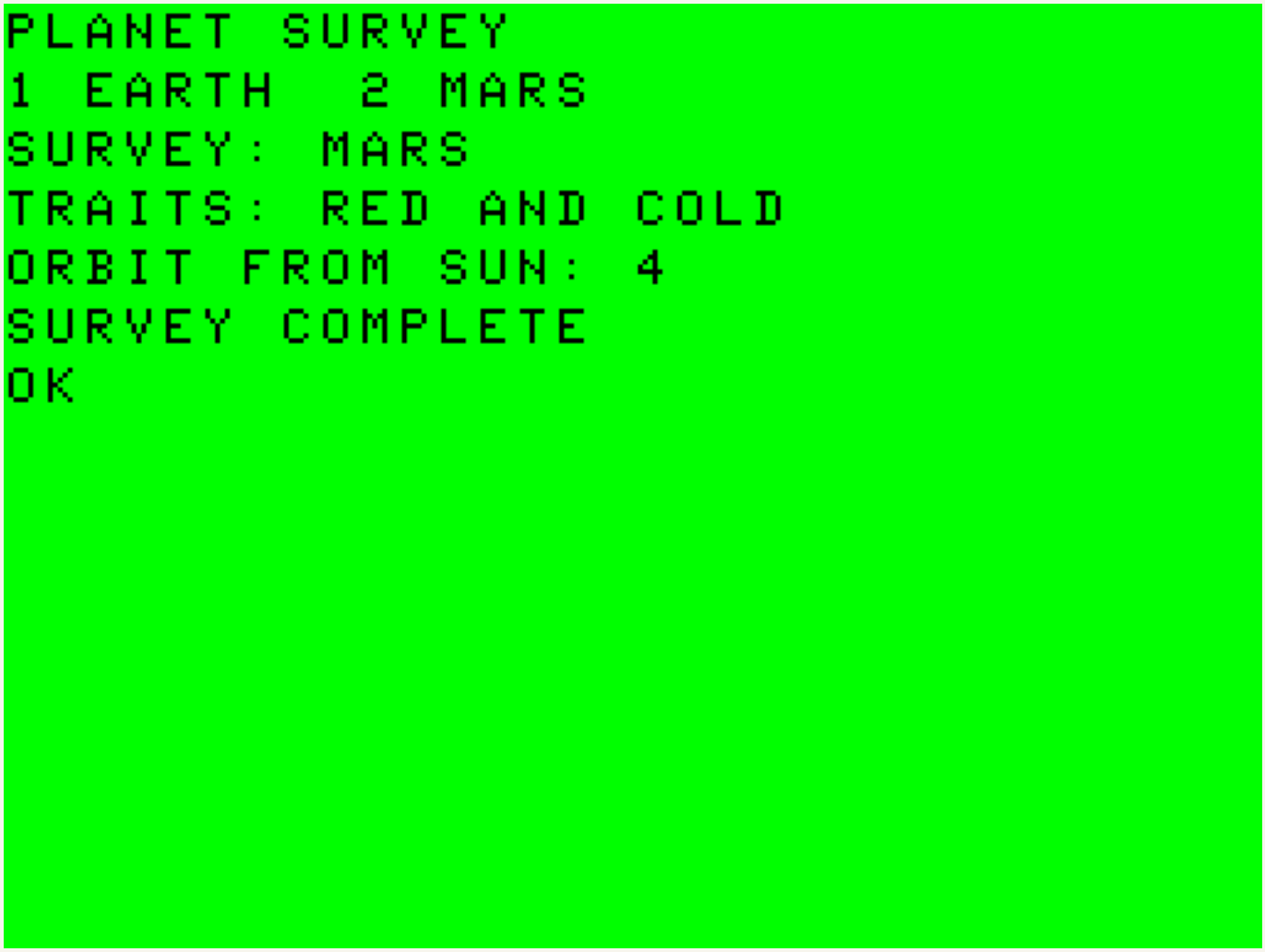


Figure 10: Planet Survey displays the two Mars traits and its orbit number after reading one disk record.

Send a new report. Change the last line to `MARS ORBIT|DUSTY,COLD|4`. Save **assets/planets.txt**, rebuild, launch the new disk and press **2** again. Look for **MARS ORBIT** and **DUSTY**. The program can split the line in place because **fgets** gave it a writable copy in RAM. The original separators are still in the file for your next trip. For the BASIC version of reading fields from disk, see Hand the log to a second program.

Keep a field log as you fly

Two planets, two observations, one disk file. Can another reader see your Earth report while your flight is still in progress?

1. Use the **Disk BASIC** profile. Save a copy of the initialized practice disk as **Field Trip.dsk**. In **Media → Floppy Disks…**, mount that copy in **FD0**. Under **Disk options** check that the row says **Saves automatically**. Each run of the original example replaces **FIELD.LOG** on the disk in drive 0.
2. Create a **Hello World** project in **Develop → C Development…**. Replace its **main.c** with fieldlog.c. Under **Project Settings → Libraries**, add **vcccompat**, then **decbfile** to `LIBS`. Choose **Build**, then **Run**. The binary runs in RAM and uses the disk you already mounted.
3. When the screen says **CHECKPOINT READER SEES**, look for **EARTH: BLUE SEA**. Your program still has its file open for writing. It opened a second stream to read that first line back from disk.
4. Click the CoCo display and press **Space**. Look under **FINAL READER SEES**. You should now have **EARTH: BLUE SEA** and **MARS: RED RIDGE**. After **SURVEY FILED**, BASIC accepts commands again.

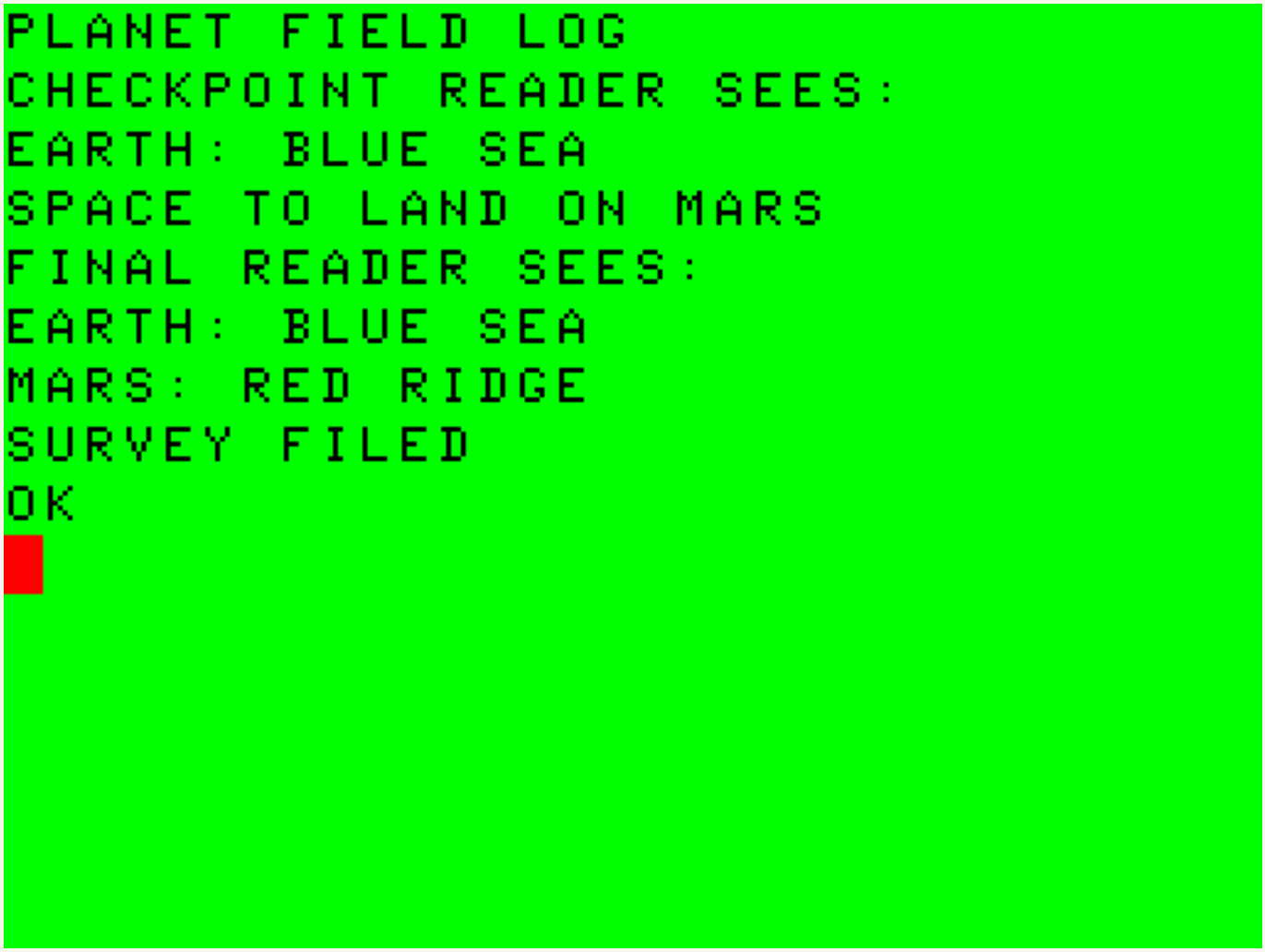


Figure 11: The running program reads the Earth report at the checkpoint, then reads both the Earth and Mars reports after closing the writer.

Find the two disk operations in the code that saves the first observation. fputs accepts the line and fflush publishes it.

```
    /* Publish the first observation while we still hold the writer. */
    ok = fputs("EARTH: BLUE SEA\n", writer) != EOF && fflush(writer) == 0;
    if (ok)
    {
        printf("CHECKPOINT READER SEES:\n");
        ok = showLog();
    }
```

A successful **fputs** means the stream accepted the characters. It can hold them in a sector buffer. A successful **fflush** writes pending output and updates the file length on the guest disk. At that point **showLog** can open its own reader and see the complete first line. Its reader starts at byte zero. Closing that reader leaves the writer ready for the Mars report. At the end, the program checks **fclose** on the writer because this last flush can also report a disk error.

Keep yesterday's observations. Change the opening mode in **main.c** from `"w"` to `"a"`. Save, build and run again with the same **Field Trip.dsk** in FD0. At the checkpoint, count the reports. Yesterday's Earth and Mars come first, followed by today's Earth. Landing on Mars adds the fourth observation. Text mode stores each C newline as a Disk BASIC carriage return. A fresh text reader presents those line endings as newlines again.

Try a protected landing. At BASIC's **OK**, switch on **Write protected** for your practice disk and run the program again. Check the error on the screen. The previous log remains on disk. With **Temporary changes** enabled instead, a successful guest flush updates the disk in the emulator. Use **Save As...** in Floppy Disks to keep a separate copy on your Mac. Compare the floppy save modes.

Read your C file from BASIC. Back at **OK**, enter these lines and type **RUN**. The same **FIELD.LOG** is now supplying strings to a BASIC program. The first two reports should be Earth and Mars. You can also leave the practice disk write-protected for this read.

```
10 CLS:OPEN "I",#1,"FIELD.LOG"
20 LINE INPUT #1,A$:PRINT A$
30 LINE INPUT #1,A$:PRINT A$
40 CLOSE #1
```

Read yesterday's log and add a return trip

Suppose you land on Mars tomorrow. How will your program remember that you already visited Earth? Leave yourself a numbered entry on the disk.

1. Use **Disk BASIC** and mount a writable copy of the blank practice disk in **FD0**. Keep that disk mounted between runs. The program keeps its travel log in **TRIP.LOG**.
2. Replace **main.c** in your **Hello World** project with returntrip.c. In **Project Settings → Libraries**, select **vcccompat**, then **decbfile**. Choose **Build** and **Run**. On an empty log, the first trip records **VISIT 1: EARTH**.
3. For a protected landing, try the save-and-retry exercise below before you fly again. For an ordinary landing, leave the disk writable and choose **Run** again. The program reads the Earth entry and seeks back to byte zero. Predict where the Mars entry will go. When the program reopens the log, you should see **VISIT 1: EARTH** followed by **VISIT 2: MARS**.
4. Run it a third time. Both visits are already on disk. The program shows them again and adds no further entry.

Look for the `"a+"` in fopen. On the CoCo it opens the log at byte zero for reading, creating the file if needed. The loop has already read to the end. The deliberate seek below puts the cursor back at byte zero so you can see what append mode does. Look for **AT BYTE ZERO** and **FILING MARS** on your second trip.

```
    /* Move the cursor to the first record. With a+, output still appends. */
    if (fseek(log, 0L, SEEK_SET) != 0)
        return 0;
    planet = visits ? "MARS" : "EARTH";
    printf("AT BYTE ZERO\nFILING %s\n", planet);
    if (fprintf(log, "VISIT %u: %s\n", visits + 1, planet) < 0)
        return 0;
```

Even with the cursor at the beginning, the write goes to the end. fprintf sends the visit number, planet name and newline straight to the stream. Once the stream has accepted the complete line, the program calls fflush to send its pending bytes to the disk. It also checks the final close.

Which kind of trip? Use **a+** to preserve the earlier visits and add a new one. Choose **w+** when you intend to start a fresh log and read back what you write. Opening with **w+** immediately empties an existing log. To change a particular byte in a file you already have, try **r+** in the atlas exercise below.

Recover the Mars entry after a refused save

Catch the second flight between writing a line in guest RAM and saving it to disk. Start just after the first run of Return Trip, with **VISIT 1: EARTH** saved in **TRIP.LOG**. If both planets are already in the log, mount a fresh writable practice disk and run the program once to record Earth.

1. At BASIC's **OK**, open **Media → Floppy Disks...**. In the **FD0** row, open **Disk options**, turn on **Write protected**, and leave this disk in the drive. Run the C project again.
2. Look for **FILING MARS** followed by **SAVE FAILED**. The stream accepted the Mars line in guest RAM. Its flush encountered the protected disk. With protection still on, click the CoCo display and press **R**. The retry reaches the same prompt. Earth is still the only visit on disk. You can press **Q** here to abandon this save, or continue to the next step.
3. In FD0's **Disk options**, turn **Write protected** off for that same disk. Return to the CoCo display and press **R** once. Under **REOPENED TRIP.LOG**, read both lines. You should see **VISIT 1: EARTH** and exactly one **VISIT 2: MARS**.
4. Run once more. **BOTH VISITS ALREADY FILED** tells you the program found its two visits on disk.

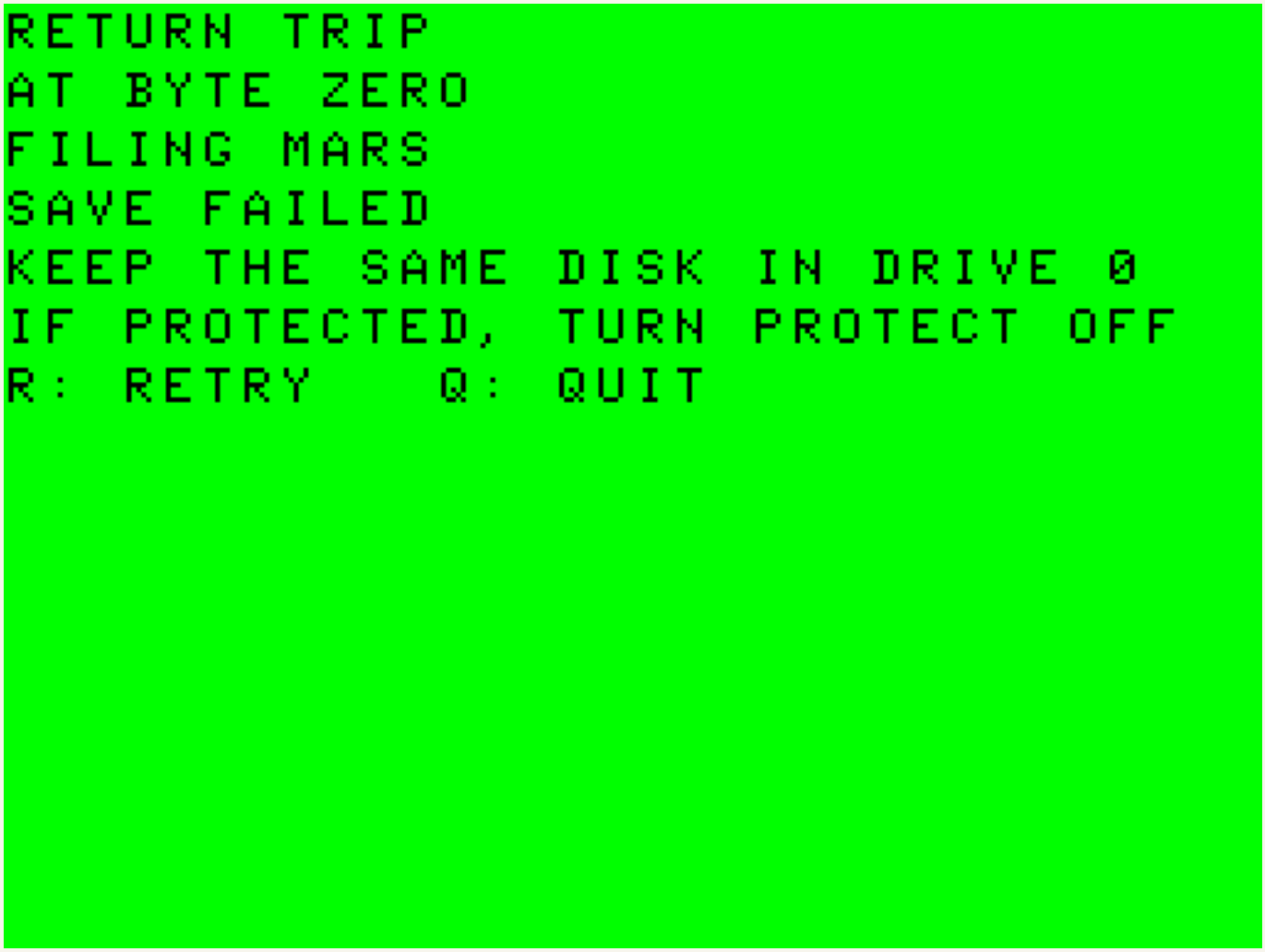


Figure 12: The running CoCo program pauses with the pending Mars save and offers a retry or quit key.

What does R repeat? It calls clearerr on the open stream, then tries fflush again. The cursor stays put and the pending bytes still contain Mars. Calling **fprintf** a second time would add a second copy of the line to the output. The recovery loop retries only the flush. You can follow the whole loop in the downloaded program. The key is read once each time the disk refuses the flush.

```
    while (fflush(log) == EOF)
    {
        char key;
        if (ferror(log) != DECB_ERR_IO)
            return 0;
```

```
        printf("SAVE FAILED\nKEEP THE SAME DISK IN DRIVE 0\n");
        printf("IF PROTECTED, TURN PROTECT OFF\nR: RETRY   Q: QUIT\n");
        do {
            key = waitkey(0);
        } while (key != 'R' && key != 'r' && key != 'Q' && key != 'q');
        if (key == 'Q' || key == 'q')
            return 0;
        /* Retry just the flush. Printing the observation again could duplicate it. */
        clearerr(log);
        printf("RETRYING SAVE\n");
    }
```

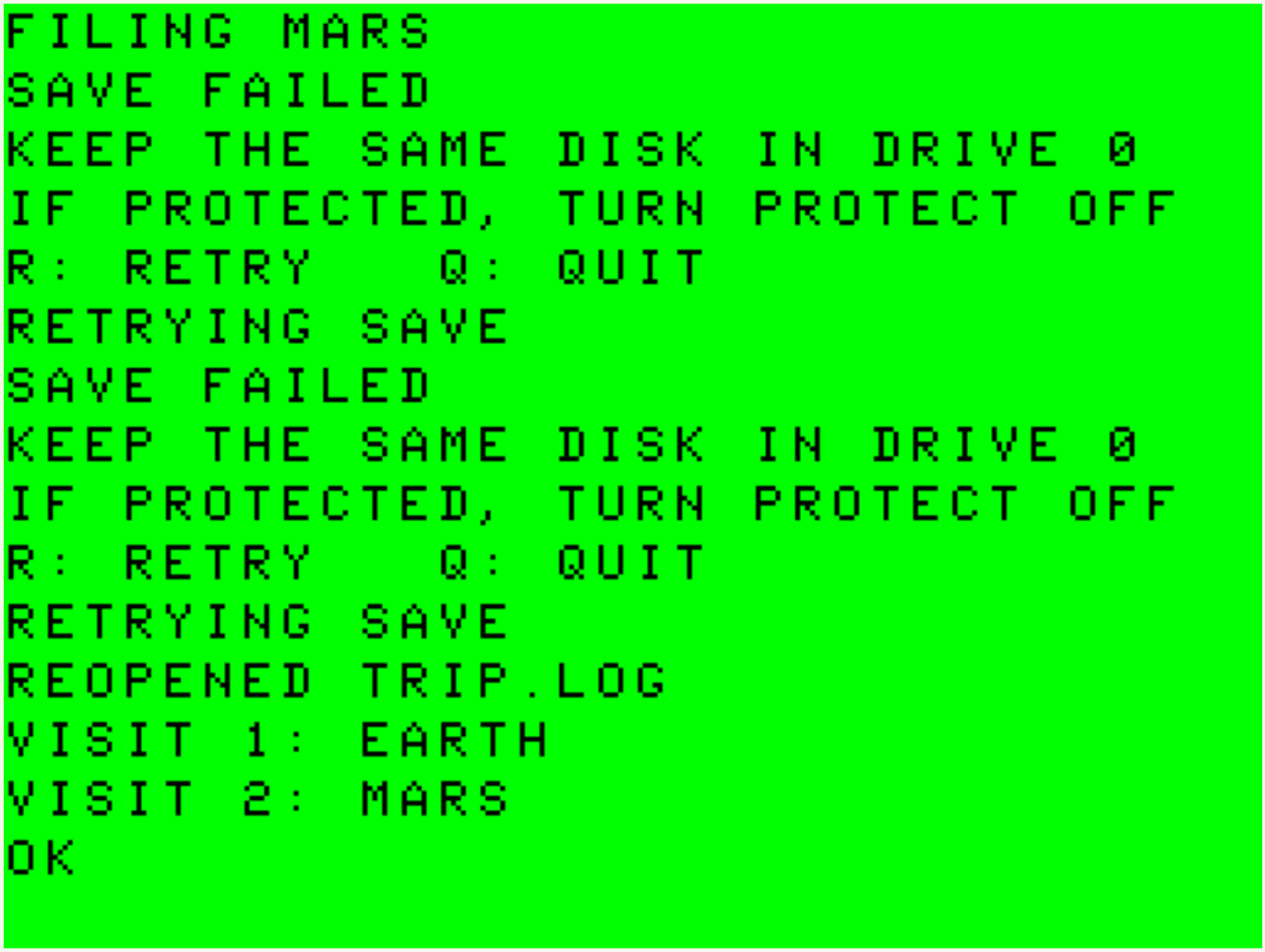


Figure 13: After protection was removed, the program reopened TRIP.LOG and displayed one visit to Earth and one to Mars.

Send a flight report to the TP-10

The crew needs its next instruction on the CoCo display. Mission Control wants a copy of the observations. You can give each one a different destination. This flight carries two observations in its C source, so you can try it with **Run** in the **Disk BASIC** profile.

1. Create a **Hello World** C project. Replace **main.c** with flightreport.c and select **vcccompat** under **Project Settings → Libraries**. Choose **Build**, then **Run**. The CoCo waits for Space.

2. Open **Media → Printer Output...**, or click the **TP-10** behind the cassette recorder in System view. Choose **Start Capture**. Click the CoCo display and press **Space** once.
3. Read **REPORT SENT TO PRINTER PORT** on the CoCo. Find the flight report in **Printer Output**. Choose **Stop Capture**. **Save Text...** keeps a readable copy on your Mac. **Print...** opens the Mac print dialog.

```
TP-10 FLIGHT REPORT
01 EARTH  BLUE SEA
02 MARS   RED RIDGE
END OF REPORT
```

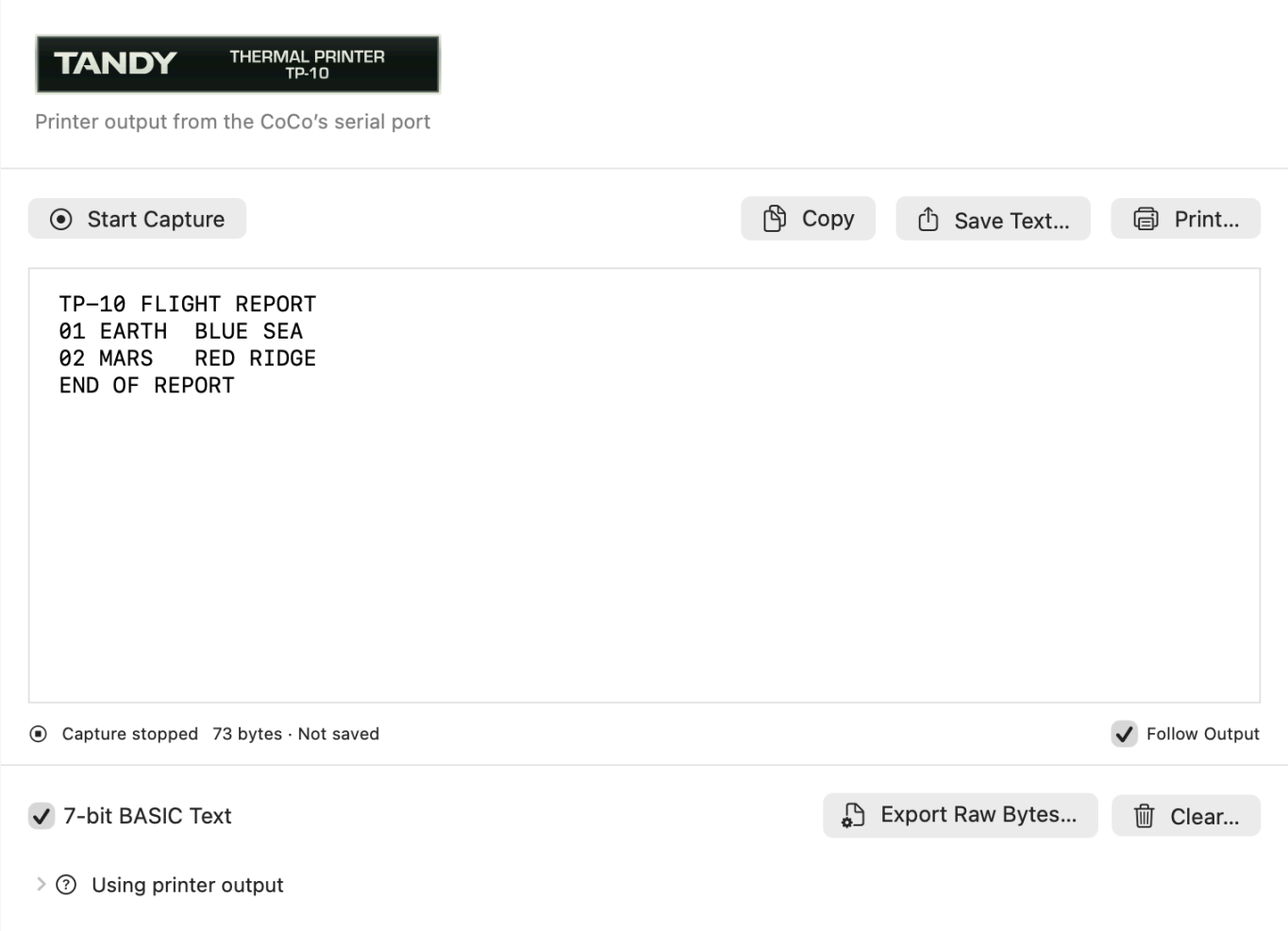


Figure 14: Printer Output retains the four-line flight report after Stop Capture. Use Save Text to keep it.

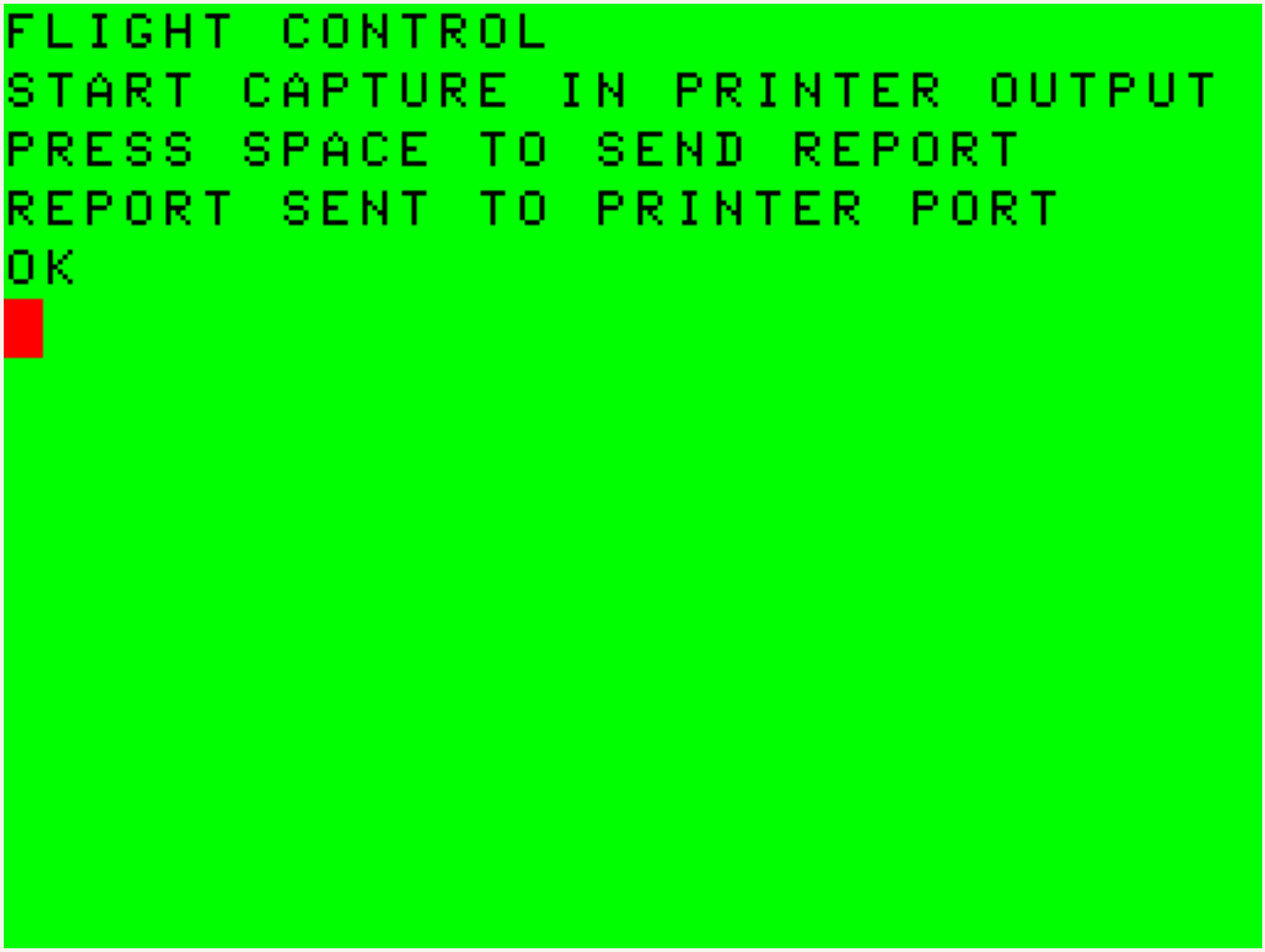


Figure 15: The CoCo shows the flight's completion message and returns to BASIC after sending the report to its printer.

Look at the two kinds of printing in the source. Ordinary **printf** tells the crew when to press Space. vcc_printer_printf formats the observations and hands them to the serial printer. Its newline ends the printed line. The ROM's printer connection carries characters one way. **REPORT SENT TO PRINTER PORT** can therefore appear even when capture is off. Turn capture on and run the flight again if you missed it.

The same helper prints both landing rows. **%02u** gives visit 1 its leading zero. **%-5s** reserves five columns for the planet and puts its name against the left edge. Earth fills the field. Mars leaves one extra space before the observation. Try a second flight with a different observation and compare your two reports.

```
static int printLanding(unsigned visit, const char *planet, const char *observation)
{
    /* Two digits for the visit, a five-character field for the planet. */
    return vcc_printer_printf("%02u %-5s  %s\n", visit, planet, observation);
}
```

For a tour of the capture window, raw byte export, and the corresponding BASIC printer command, see Book 5's printer receipt.

Exchange a flight recording with BASIC

The TP-10 gave Mission Control a printed report. This time a second program will read the observations. Give each observation one line in a cassette file named **FLIGHT**. A reader finds Earth first and Mars second. To read Earth again, you rewind the tape.

Record in C, read in BASIC. Use the **Disk BASIC** profile. Save any current BASIC program and export any cassette recording you want to keep before you begin.

1. Create a **Hello World** C project. Replace **main.c** with Flight Recorder. Select **vcccompat** under **Project Settings → Libraries**, then **Build** and **Run**. Leave the CoCo waiting at **W WRITE R READ**.
2. In **Media → Cassette…**, choose **New Recording → CAS**. Click the CoCo display and press **W**. Wait for **FLIGHT SAVED** and the BASIC prompt. In the recorder choose **Stop**, then **Export Tape…**. Save it as **First Flight.cas**. The Mac filename identifies the tape image. **FLIGHT** is the name the two guest programs use to find the data on it.
3. Choose **Rewind**, then **Play**. Open the BASIC reader in BASIC Development and run it. You should see **EARTH BLUE SEA**, **MARS RED RIDGE**, then **END OF FLIGHT**.

```
10 OPEN "I",#-1,"FLIGHT"
20 IF EOF(-1) THEN 60
30 LINE INPUT #-1,A$
40 PRINT A$
50 GOTO 20
60 CLOSE #-1
70 PRINT "END OF FLIGHT"
```

BASIC device **#-1** is the cassette. Line 20 checks for the end before reading. Line 30 takes a whole observation, including its spaces. Line 50 goes back for the next one. Your C writer can change the number of observations without changing this reader.

Record in BASIC, read in C. The return flight sends different observations so you can tell which tape supplied them.

1. Open the BASIC writer. Choose **New Recording → CAS** and run the writer. Wait for **FLIGHT SAVED** and **OK**, then stop and export the tape with a second name, such as **Return Flight.cas**.
2. Choose **Rewind**, then **Play**. In C Development, run **Flight Recorder** again. Click the CoCo display and press **R**. Look for **EARTH TIDE POOL** followed by **MARS DUST DEVIL** and **END OF FLIGHT**.

```
10 OPEN "O",#-1,"FLIGHT"
20 PRINT #-1,"EARTH TIDE POOL"
30 PRINT #-1,"MARS DUST DEVIL"
40 CLOSE #-1
50 PRINT "FLIGHT SAVED"
```



Figure 16: Flight Recorder in C displays EARTH TIDE POOL and MARS DUST DEVIL from the BASIC recording, reaches END OF FLIGHT and returns to OK.

Opening a file for output records its name. Each line adds a record. Closing the file records its ending. If a writer opens **FLIGHT** and immediately closes it, the reader should reach **END OF FLIGHT** without printing an observation. If the recorded ending is missing or a block is damaged, Flight Recorder reports **TAPE ERROR**. A recording can contain both observations and still be incomplete. On the same cut-short tape, the BASIC reader can sit at the end waiting for another block. A block with a bad checksum can produce **?IO ERROR**. For a fresh attempt, use your complete exported recording.

The C reader gives each line a 32-byte array, including space for the final zero. Check the return value before printing that array. Look up the cassette functions, tour the recorder controls in Book 5, or compare tape order with seeking to a planet on disk.

Jump to a planet in the atlas

Imagine keeping the planets on four index cards. If every card takes up the same amount of space, you can jump straight to Mars. In **Planet Atlas**, each card occupies 24 bytes of a binary disk file. Mercury, Venus, Earth and Mars start at bytes 0, 24, 48 and 72.

1. Mount a copy of the initialized practice disk in **FD0** on the **Disk BASIC** profile. Leave the disk writable. On a fresh disk Planet Atlas creates **ATLAS.DAT** and writes the four cards once. On later trips it opens the existing atlas with **r+b**, so it can read a card and mark your visit.

2. In **Develop → C Development…**, create a **Hello World** project. Replace **main.c** with atlas.c. Under **Project Settings → Libraries**, add **vcccompat**, then **decbfile** to LIBS. Choose **Build**, then **Run**.
3. Click the CoCo display, type **4** and press **Enter**. Look for **YOUR PICK AT BYTE 72** and **MARS: RED RIDGE**. The program visits Earth, returns to the bookmarked Mars card and rewinds to Mercury. Look for **VISIT SAVED** after the first trip to Mars. At **OK** you are back in BASIC.



Figure 17: The CoCo shows Mars at byte 72, rewinds to Mercury, then reports VISIT SAVED and ATLAS CLOSED before BASIC's OK prompt.

The first time through, the program writes four records, padding the unused bytes of each with zeros. The first record is number 1 but its first byte is at offset 0. Subtracting one before multiplying by the card size accounts for that difference.

```
    /* Card numbers begin at 1. Disk offsets begin at 0. */
    long offset = (long)(number - 1) * CARD_SIZE;
    long bookmark;
    if (fseek(atlas, offset, SEEK_SET) != 0)
        return 0;
    bookmark = ftell(atlas);        // Save the start before reading the card.
    if (bookmark < 0)
        return 0;
    printf("YOUR PICK AT BYTE %ld\n", bookmark);
```

fseek moves to the chosen byte. **SEEK_SET** measures from the start of the file. ftell gives you a position you can keep in a **long**, which is 32 bits in CMOC. Reading a card advances the stream by 24 bytes. If you saved the position *after* reading Mars, where would the bookmark take you? Byte 96 is the end of this atlas. Saving the position before the read lets you find Mars again after the side trip to Earth.

rewind returns to byte 0 so the program can read Mercury.

Binary mode makes these offsets easy to calculate. In a text stream a line ending can occupy more than one byte. When you want to return to a place in text, save a bookmark from **ftell** and pass it back to **fseek** with **SEEK_SET**.

Leave a mark for your next visit

Run the same program again with the same practice disk in drive 0. Choose **4** for Mars. This time look for **ALREADY VISITED**. Your program just started over, so where did it find that memory?

Each card uses the last of its 24 bytes for the visit mark. On a fresh card that byte is zero. The Mars card starts at byte 72, so its mark lives at byte 95. On the first visit the program reads the card, seeks to the mark and writes the single character **V**. It calls fflush to publish that byte to the disk. Next time, reading the same card brings the **V** back. Earth's card at byte 48 still has its own mark. Showing Earth as home base did not mark Earth as your selected destination.

```
    // Position the update stream between reading the card and writing its flag.
    if (fseek(atlas, start + CARD_SIZE - 1, SEEK_SET) != 0 ||
        fputc('V', atlas) == EOF || fflush(atlas) == EOF)
        return 0;
    printf("VISIT SAVED\n");
```

If you edited the atlas on an earlier trip, a full-length description might use that last byte. **NO ROOM FOR VISIT MARK** means the program kept your card intact.

With an update stream the same file handle reads and writes. The stream remembers *one* position for both operations. After reading the entire Mars card the position is already 96. Seeking to byte 95 tells the stream where to write and permits the change of direction. After writing, a successful flush or seek permits reading again. Check each result before claiming a visit was saved.

Try refusing a visit. Wait for BASIC's **OK**, turn on write protection for your practice disk, and run the atlas again. Choose **2** for Venus if you have only selected Mars so far. Look for **CANNOT SAVE VISIT**. Choosing Mars still works because its **V** is already there. Restore the disk to writable for the next exercise.

Which answers reach Venus?

1. Choose **Run** again. Type **2X** and press **Enter**. The atlas says **EXTRA CHARACTER: X** and asks again. Can you see why a program that only checks the numeric result would send you to Venus?
2. Type **0x2** and press **Enter**. Look for **YOUR PICK AT BYTE 24** and **VENUS: CLOUDS**. The program reads your answer in hexadecimal.
3. Run once more. Type **2** and press **Enter**. This time your answer is decimal. Both spellings select the same card and set the same bookmark.

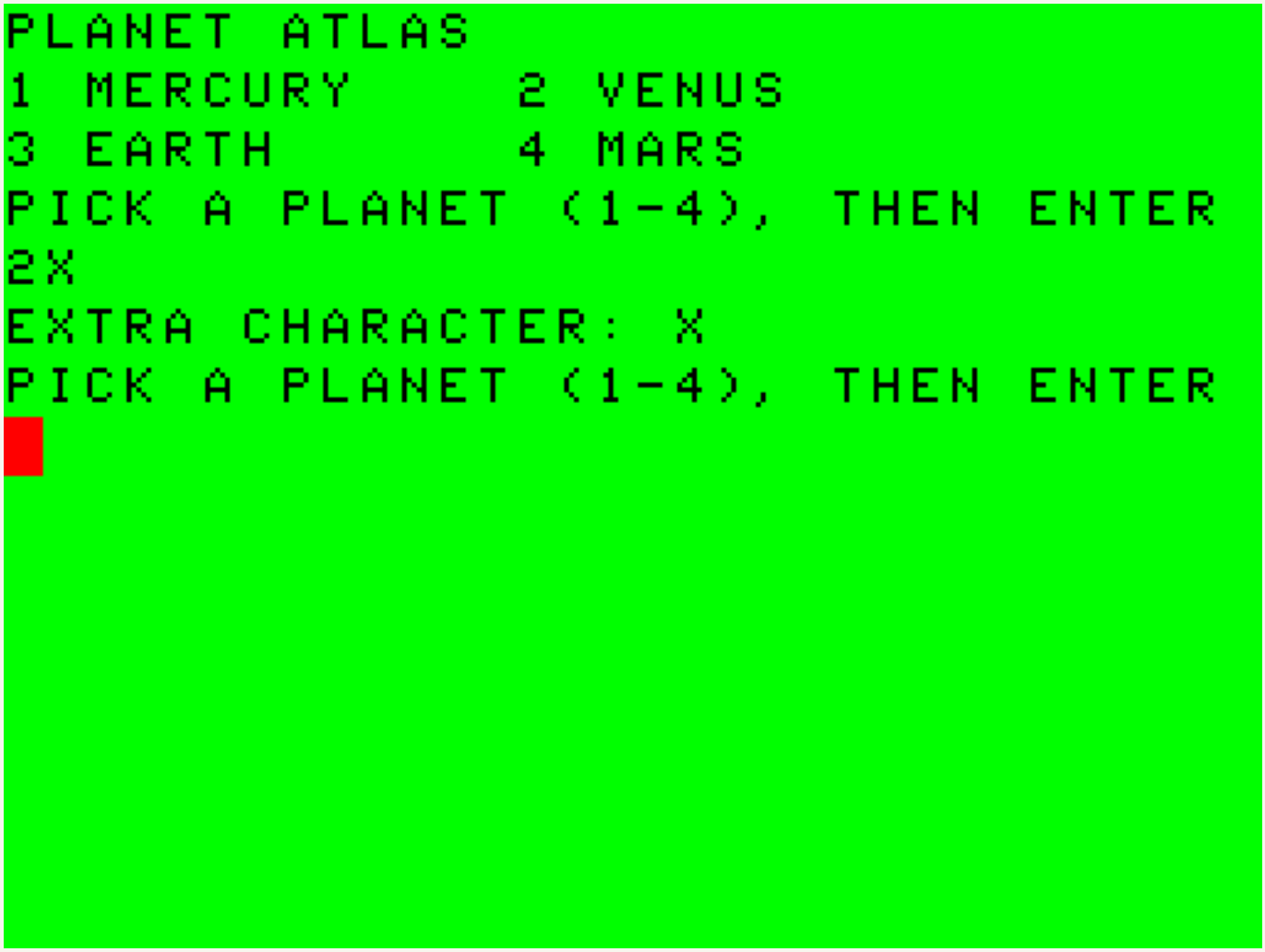


Figure 18: Planet Atlas identifies the X in 2X and asks for another destination on the guest screen.

Look inside **choosePlanet**. It asks for a line beginning with a digit. Passing zero for the base tells strtoul to recognize prefixes. Then the function leaves **end** pointing into the line. For **2X**, it points straight to the **X**. For **0x2**, it reaches the zero byte that marks the end of the string.

```
        errno = 0;
        number = strtoul(line, &end, 0);
        if (*end)
            printf("EXTRA CHARACTER: %c\n", *end);
        else if (errno == ERANGE || number < 1UL || number > 4UL)
            printf("CHOOSE A NUMBER FROM 1 TO 4\n");
        else
            return (int)number;  // Now the value is small enough for an int.
```

There are two limits here. The conversion must fit in an unsigned 32-bit **long**. Clearing errno first lets you check for **ERANGE** on this attempt. The atlas itself has just four cards. It checks both limits before putting the result into a smaller **int**. For input that can be negative, see the signed companion, strtol.

Rescue an interrupted atlas save

The Mars survey team has a new observation: **MARS: DUST STORM**. You want the new atlas on disk, but what if the CoCo restarts halfway through replacing the old one? With three directory names, you can leave the next crew enough evidence to recover.

Name on drive 0	What it holds
ATLAS.DAT	The atlas readers open.
ATLAS.NEW	The proposed new edition. The program closes it, reads it back and compares every byte before moving the original.
ATLAS.BAK	The previous complete edition, including the visit marks.

1. Mount a **writable copy** of the blank practice disk in FD0 under **Disk BASIC**. Use automatic saving. You can also use a copy of the disk from the four-card atlas, including its **V** marks. Save other work in guest RAM before trying the restart.
2. Create a **Hello World** C project, replace **main.c** with atlasbackup.c, and select **vcccompat** followed by **decbfile** in **Project Settings → Libraries**. Build and Run. On a blank disk the program first creates the original four cards. Wait for **BACKUP IN PLACE** and **PRESS A KEY TO INSTALL**. Leave the keyboard alone for a moment. Which two filenames would you expect in a directory right now?
3. Use the emulator toolbar's **Restart → Cold Restart — Clear RAM…** and confirm. Keep the same practice disk mounted. At BASIC's **OK**, type **DIR 0**. You should find **ATLAS BAK** and **ATLAS NEW**. Both editions exist. At this instant neither has the name **ATLAS.DAT**.
4. Run the same project again. Look for **ORIGINAL ATLAS RESTORED** and **MARS: RED RIDGE**. Type **DIR 0** after the program returns to BASIC. **ATLAS.DAT** is back. The program removed its verified draft.
5. For comparison, run it again and this time press **Space** at **PRESS A KEY TO INSTALL**. Look for **UPDATED ATLAS SAVED** and **MARS: DUST STORM**. **DIR 0** now shows **ATLAS.DAT** and **ATLAS.BAK**. Running again reports **ATLAS ALREADY UPDATED**. The backup still holds the old ridge.



Figure 19: Planet Atlas Backup in a freshly started CoCo session reports ORIGINAL ATLAS RESTORED and reads MARS: RED RIDGE from the recovered edition.

Each rename changes one name on the same disk. Here's the point where you interrupted the pair:

```
    if (rename(LIVE, BACKUP) != 0)
        return 0;
    printf("BACKUP IN PLACE\nPRESS A KEY TO INSTALL\n");
    waitkey(0);
    if (rename(DRAFT, LIVE) != 0)
        return 0;
```

When the program starts again, it reads the directory and the cards. Finding the original edition under **ATLAS.BAK** with **ATLAS.DAT** absent is its cue to restore that name. Then it calls remove only if **ATLAS.NEW** contains exactly the revision it would have made, visit marks and all. The code leaves unexpected contents for you to inspect. A custom atlas with extra cards or different descriptions belongs on its own disk.

The promise you tested is precise: restarting **between two completed renames** leaves a complete atlas to recover. Disk BASIC has no journal to repair a disk damaged during a sector write. Keep the original Mac disk image when experimenting on a copy.

Land on a second disk without leaving your program

You're flying an expedition from the **CoCo SDC** profile. Drive 0 holds the flight program. Drive 1 is your destination. Earth and Mars each carry a file named **LANDING.TXT**. Can the same line of C find two different sets of landing instructions?

1. Save the Planet Expedition project, expand it, and open **PlanetExpedition** in **Develop → C Development…**. Choose **Build**. Under **Build and file actions (…) → Show Build Products**, find **EXPED**. Copy that whole folder to a new location in Finder. The copy is your practice card.
2. Save work in guest RAM before changing profiles. Load **CoCo SDC** from **Machine → Configuration Profiles…**. Under **Cartridge → Cartridges…**, open the CoCo SDC details. Note your current card folder so you can return to it. Use **Choose Folder…** to select the copied **EXPED** folder. You can enable **Protect SD card files (read-only)** for the entire trip.
3. At the **SDC-DOS** prompt, type these commands. The first places the program disk in SDC0. The second loads and starts the program saved on that disk.

```
DRIVE 0,"/EXPLORE.DSK"
LOADM"EXPLORE:0":EXEC
```

The chart lists the disk images it finds in your card folder. Press **1** for Earth. Look for **SPLASH DOWN BESIDE THE BUOY.** Now press **2** for Mars. How did **LAND BEYOND THE DUST FIELD.** arrive on the same screen?

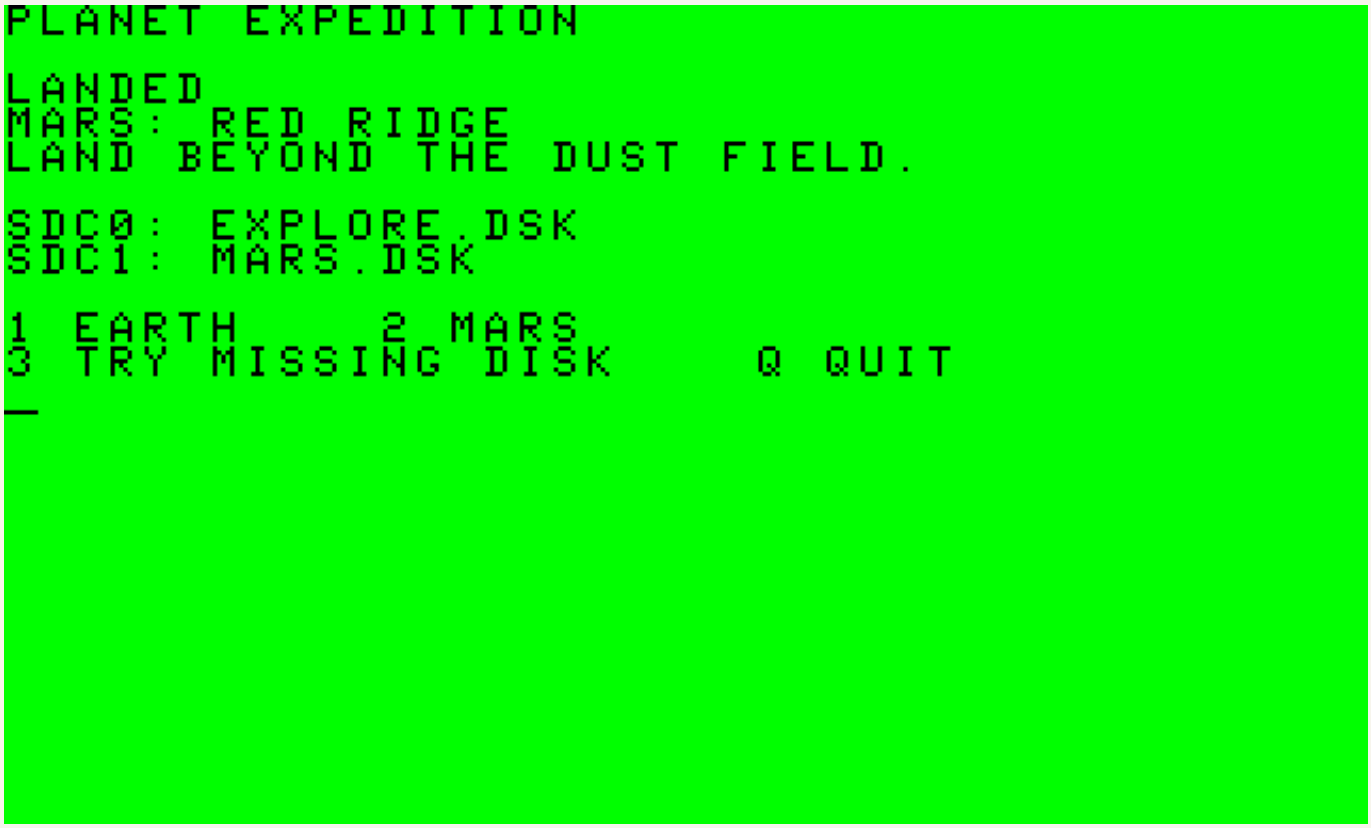


Figure 20: The running CoCo shows the Mars landing instructions. EXPLORE.DSK is still mounted on SDC0 and the Mars data disk is mounted on SDC1.

Follow the two filenames in expedition.c. At startup, sdc_list_begin and **sdc_list_next** read the outer directory on the card. When you choose Mars, sdc_mount selects **/MARS.DSK** for drive 1. The slash names a file at the

root of the card. Inside the newly mounted image, **LANDING.TXT:1** names a Disk BASIC file. Its **:1** tells the guest which drive to read.

```
static void landingReport(void)
{
    // :1 asks the Disk BASIC ROM for a file inside the mounted planet.
    FILE *file = fopen("LANDING.TXT:1", "r");
    char line[40];
    int ok;
```

The program reads the landing instructions with **fgets** and closes the stream before accepting another destination. Keeping a file open across a disk change would leave its directory and sector information pointing into the old image.

Miss a landing. With Mars in drive 1, press **3**. The practice card has no **MISSING.DSK**. Look at the error, the repeated Mars report and the two actual mounted names returned by **sdc_mounted**. A failed request let you keep flying with the same disks. Press **Q** to return to BASIC. Type **DIR 0** to see **EXPLORE BIN**, then **DIR 1** to see **LANDING TXT**. Select your original card folder when you finish.

Send your own instructions. Edit **assets/mars.txt**, rebuild, and take the new **EXPED** folder to a second practice location. Choose that card and repeat the two SDC-DOS commands. The guest should now print your new words when you select Mars.

Find the file that went missing

Chapter 7 of 9

Suppose Hello Disk builds successfully, prints the heading for the words, and immediately skips to the result. Which end needs a repair: the Mac build, the image, or the program on the CoCo? Change one thing so you can answer that question from evidence.

Make a fault you can undo

Use a private five-word **Hello Disk** project in **Debug**. In **src/disk.c**, change only the filename in the open call from **WORDS.DAT** to **WRONG.DAT**. Save All and **Build Disk**. Leave the resource list alone. The tools have every file they need on the Mac, and WRONG.DAT is a perfectly valid C string. Expect a successful build.

Catch the returned value

Find the second `if (status == DECB_OK)` in **showDiskWords**, immediately *below* the call to decb_openSectorFile. In the shipped sample it is line 18. In the **Source breakpoint** picker at the bottom of the C workspace, select **Line 18**, then click the adjacent plus button to add the stop. If you stop on the call itself, the assignment you want to measure is still ahead of you. If you stop at the previous condition, you are measuring registration of drive zero.

1. Choose **Launch Disk** so the guest uses the newly built image. At the source stop, select the **showDiskWords** frame. Open **Variables**, or enter **status** in **Watches**.
2. Read **3**. In decbfile this is **DECB_ERR_NOT_FOUND**. A successful build and a file-open failure can both be true.
3. Select the **main** frame in the call stack. It can show where the disk operation was requested. Its own variable named status receives a return value only when showDiskWords finishes. Select the inner frame again to read the meaningful value at this stop.
4. Continue. Expect **DISK RESULT: 3**, **GENERATED WORD COUNT: 5**, and no printed words. At **OK**, type **DIR**. The disk still lists **WORDS DAT**. Compare the spelling with the C string.

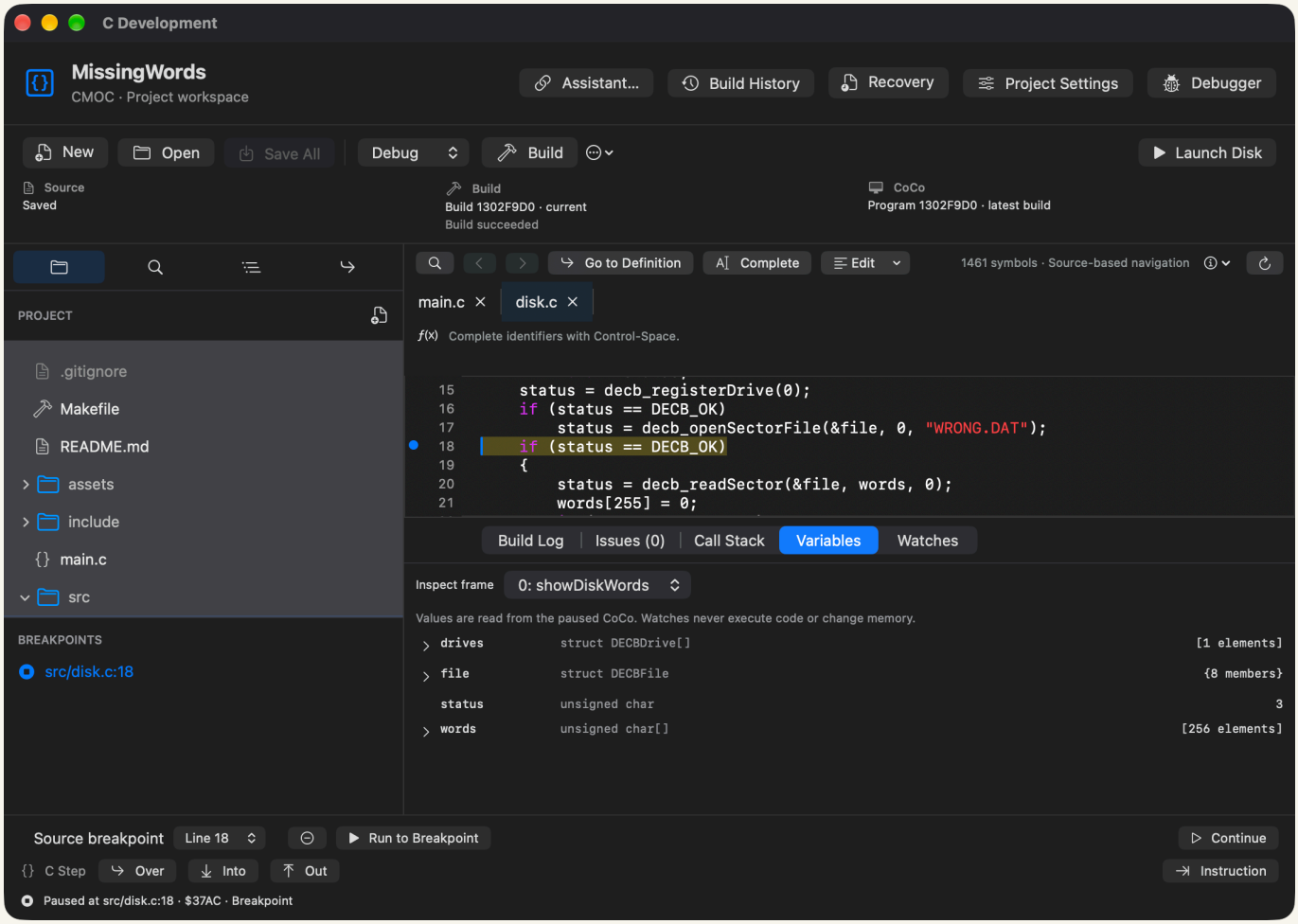


Figure 21: A private Hello Disk copy named MissingWords has stopped on line 18, immediately after the open of WRONG.DAT. In the showDiskWords frame, Variables reports status 3.

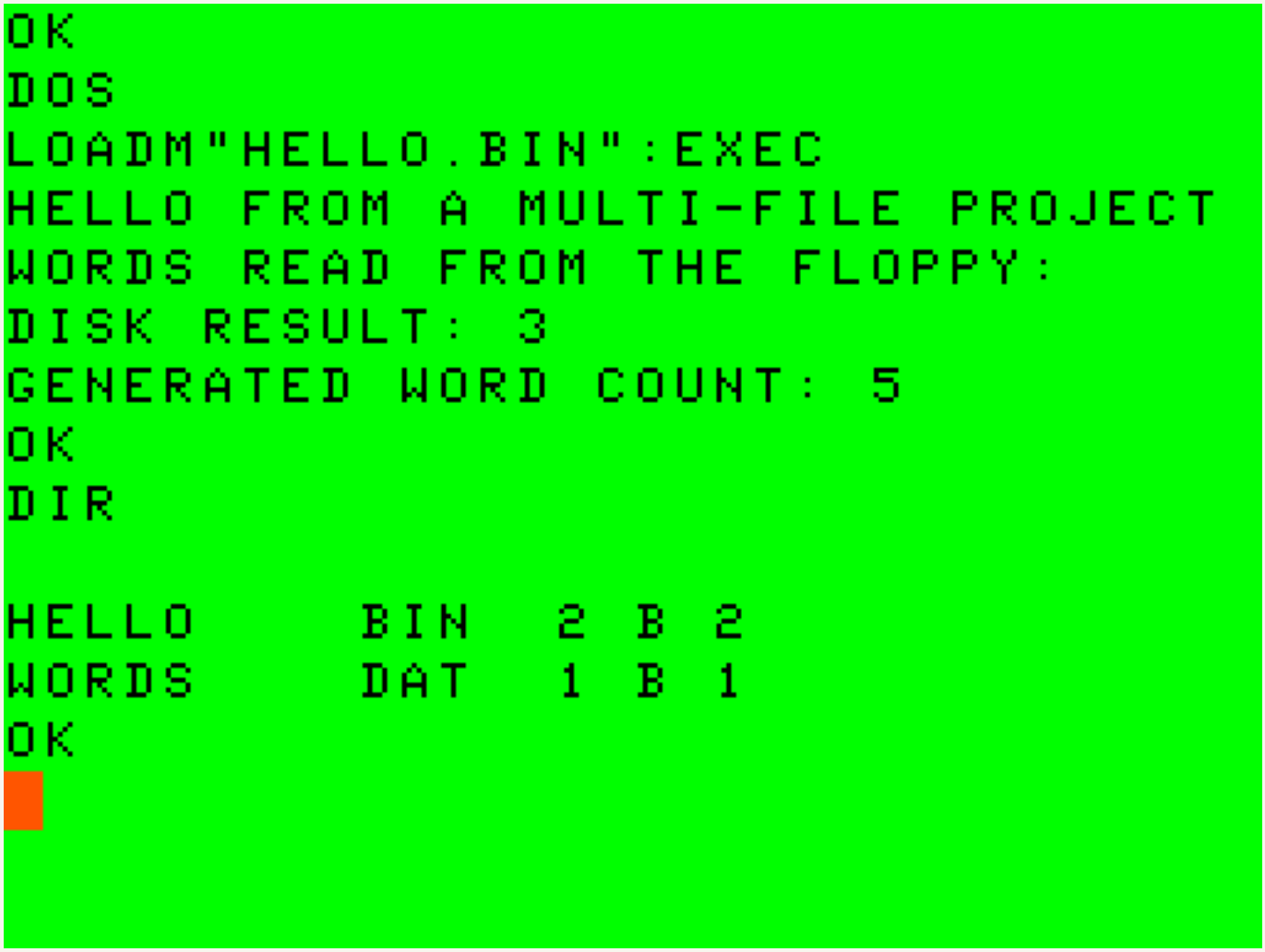


Figure 22: The same guest screen shows disk result 3 and count 5 above the DIR listing for HELLO BIN and WORDS DAT.

If the debugger says a source location is unavailable, check that you built the saved Debug sources and launched the matching artifact. The editor can remember a line while the executable and address behind that line change. Resolve a stop against the new build before interpreting its variables.

Repair one name

Restore **WORDS.DAT** in src/disk.c, save, build the disk and launch it. Use the same post-open source line. This time **status** is **0**, the library's DECB_OK. Continue: the five words reappear. The count still says **5** because the word list was never edited. With the breakpoint selected, click the adjacent minus button to clear it.

You have distinguished three independent claims. The linker produced an executable. The directory contained a named file. The program found that file at run time. Each was worth checking in a different place.

For a program with no floppy dependency, try a graphics, input or sound experiment. To make a copy for a friend, go straight to the six-word disk.

Choose a library for a job

Chapter 8 of 9

A landing beacon needs a mark on the screen. A pilot needs a way to steer. A landing signal needs a sound. Try one of the small experiments on its own, or open the complete Mission Recorder project to see how the graphics, input, sound, storage and printer libraries work together on one CoCo.

Put the call and its library together

Start a fresh **C Development → New → Hello World** project for your experiment. Save the linked C example from this chapter and use its contents as that project's **main.c**. Keep your existing projects in their own folders.

Open **Project Settings → Libraries → Add Library**. Select the library named in the lesson. Set **Append to linker variable** to `LIBS` and leave **Directory variable** empty. Save your settings and choose **Build**, then **Run**. The Hello World Makefile already passes `LIBS` to the final compiler command. If you use a settings-generated Makefile instead, enter the library name under **Linked libraries**, one name per line.

The names on either side of this arrangement can differ. For the sound lab the header is **Player.h** and the library is **bsound**. A missing header stops compilation. An unresolved symbol such as `_Player_sound` usually means the declaration was found and the linker still needs the library implementing it. You can look up either name in Find a call.

Try the runtime with a cargo tag

Use message.c in a fresh Hello World copy. Skip **Add Library** for this experiment. All the calls come from the runtime header `cmoc.h`.

A cargo tag has room for four visible letters. Why does the source declare a five-byte array? `strlen` counts the letters. A C string also ends with a zero byte, so `memcpy` copies one byte more than that count. Before making the copy, `setTag` compares the length with the capacity supplied by the caller. Follow both requests. **MARS** fits. **EARTH** has five letters and needs six bytes. The refusal leaves the earlier tag untouched.

The guest reports **TAG: MARS, LETTERS: 4**, then **EARTH TOO LONG. KEPT MARS**. The second line depends on `strcmp` returning zero, C's result for two equal strings. The first uses `%s` for the terminated tag and `%u` for an unsigned count.

Draw a beacon you can measure

Choose **Settings → Video Signal → RGB**. Use beacon.c with **bgraph**. Run it and look for a white cross. Its middle is where two lines of ink meet. Press a key and the program goes back to text and prints **BEACON CENTER: 3**. How can a white spot be 3?

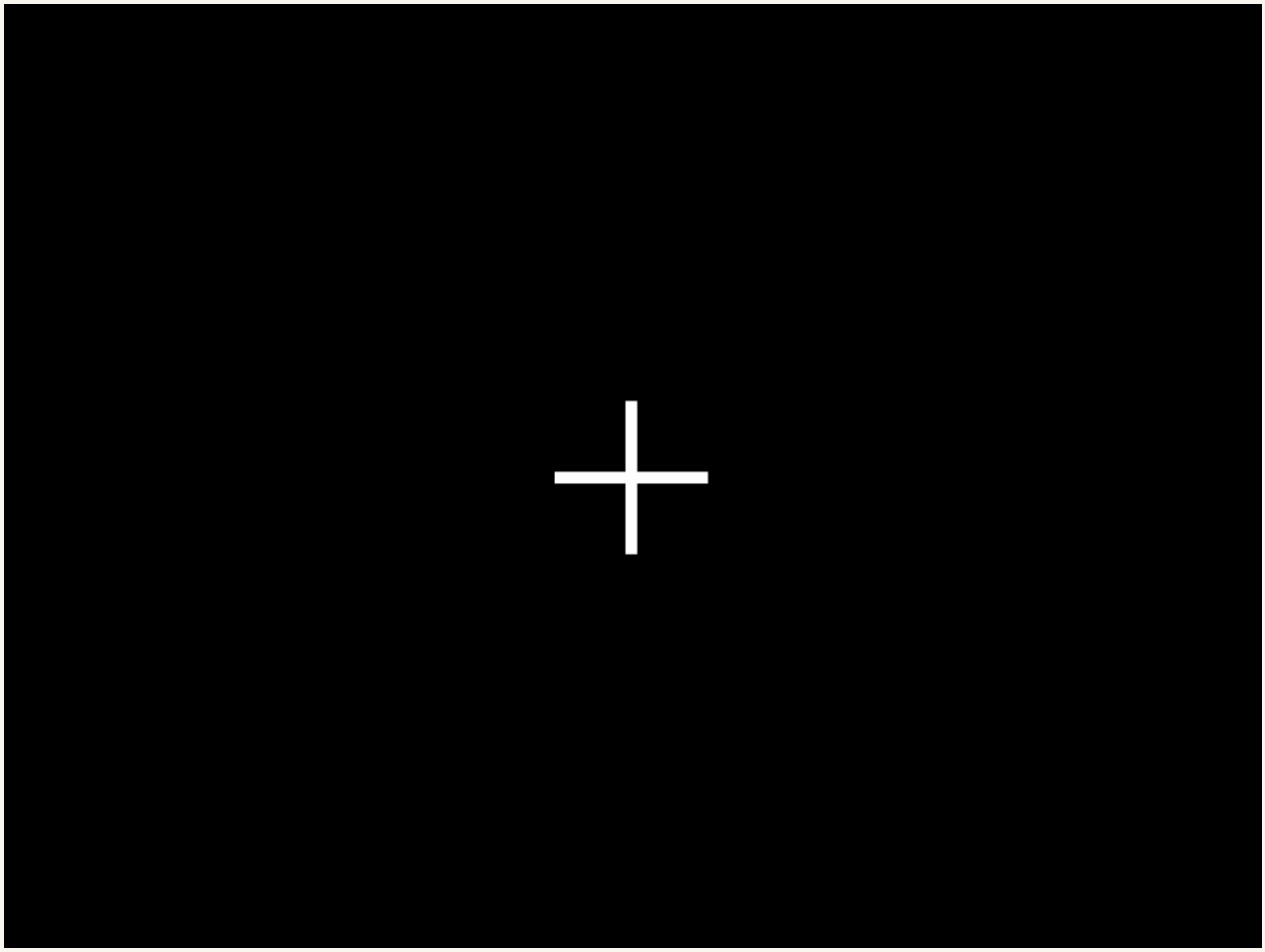


Figure 23: VCC Plus displays the white landing cross on the black CoCo 3 graphics page while the program waits for a key.

Open the source and find `INK`. The stamp holds **palette slot numbers**. The program assigns white to slot 3. In HSCREEN 2, each byte holds two four-bit pixels. A full 320-pixel row occupies 160 bytes. The program keeps just 13 rows in a static buffer, calls `Primitive_setPixel320x16` to draw the cross and calls `Primitive_getPixel320x16` to read each dot. Both functions expect the 160-byte stride even though our little stamp uses only its first 13 columns. Keep x and y inside the allocated area.

The buffer belongs to your C program. The displayed page belongs to the guest. The two inner loops use `hset` to copy each dot as a three-by-three square into the center of that page. Look back at the white cross. Its shape came from BGraph, its size from the copy loop and its brightness from the palette. A black screen can mean the program has already received its key or the foreground and background slots are the same. The last lines restore text mode and the two saved palette entries. Compare the BASIC star chart.

Catch the pilot's signal

Use stick.c with **bcontrol**. In **Input → Keyboard and Joysticks…**, choose **Keyboard** for the **Right CoCo joystick**, then choose **Arrows** under its key options. Remember the previous assignment so you can put it back. A physical controller assigned to the same port works too. See the port controls.

Run the program, give the CoCo display keyboard focus, and hold Left and Down together. Tap **Z** for fire while holding the direction keys. The program snapshots the right stick, reports **RIGHT X=0 Y=63** at full travel and prints **BUTTON DOWN**. Release the controls before typing at BASIC. Run once more with the stick centered and

your hands off. After about ten seconds it reports **RIGHT X=31 Y=31** on a centered virtual stick and **BUTTON UP**. A physical controller may settle a little to either side of center.

Read the two masks in the source. The axis mask `0x03` requests axes 0 and 1. The button mask `1` tests right Fire 1. A set button bit means **up**. BASIC's timer gives the wait a deadline, so even someone with nothing assigned can finish the experiment. The joystick and sound share hardware. The program silences the analog multiplexer for polling and reconnects the sound input when finished.

If you always get the centered result, check the live input numbers on the **right** port card before editing the C program. Movement in the Mac controller preview proves only that the Mac sees the controller.

Fly a lander with two screen pages

There is a gold landing pad over the ridge. Can you bring both feet down on it? Use a **CoCo 3 with at least 512 KiB** in the Disk BASIC profile. Choose **Settings** → **Video Signal** → **RGB** to see the colors in the exercise. Save lander.c as the **main.c** in a fresh Hello World project. Add **gime**, **bgraph** and **bcontrol** to `LIBS` as described above. Choose **Build**, then **Run**.

Click the CoCo display and steer with **A/D** for left/right and **W/S** for up/down, or use a joystick assigned to the **right CoCo port**. The lander holds its position when you release the keys and center the stick. Move right until the entire craft is above the gold bar, then descend. **ON THE PAD!** means both feet made it. **Q** ends a flight early with **FLIGHT ENDED**. Either route returns to BASIC.

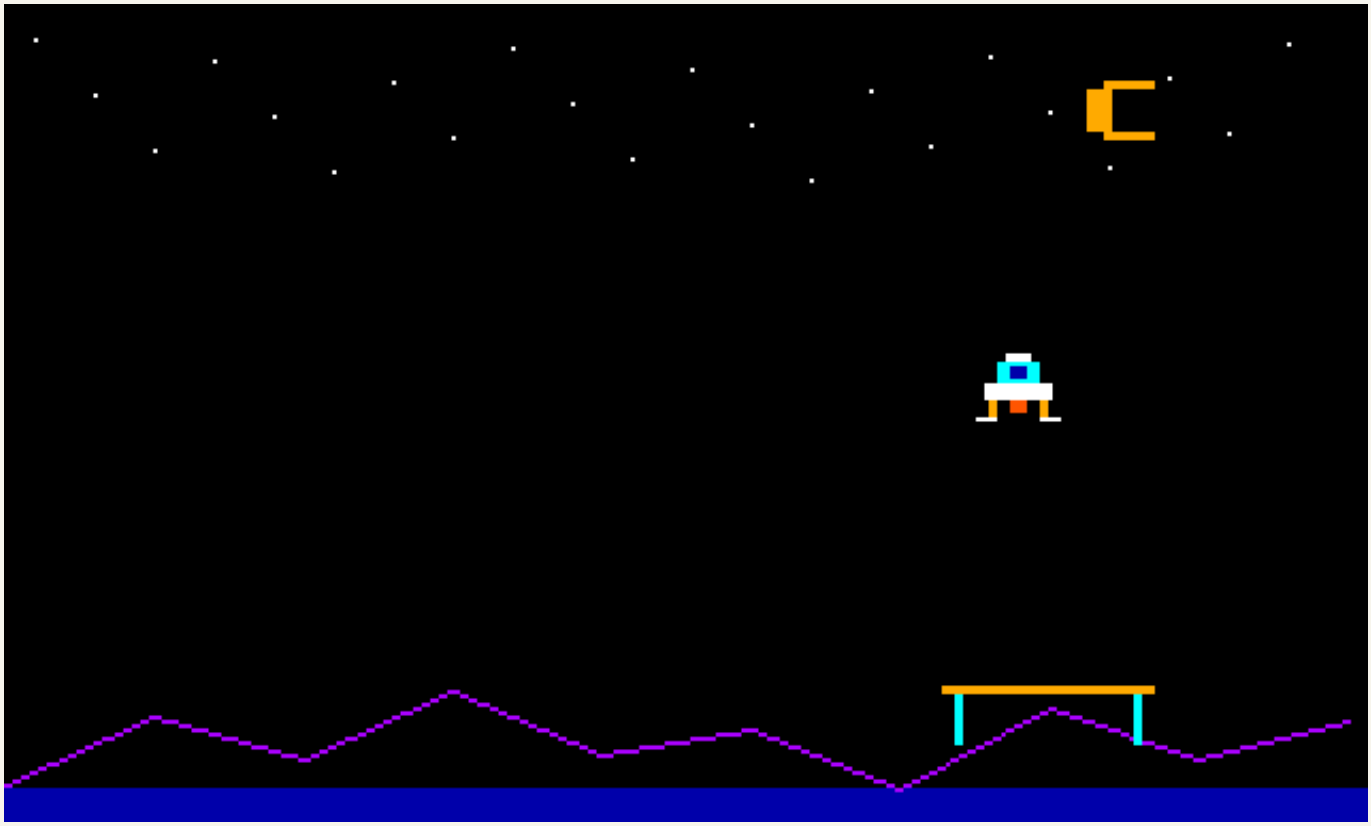


Figure 24: The running lander hovers above the gold pad on the right. The cyan legs of the pad cross a violet ridge, and a gold crescent hangs above the craft.

Follow one movement in **main.c**. While you see the lander at its last position, the GIME is reading the **front page**. The program erases the old craft and draws the new one on the **back page**. gime_present waits for a field boundary and swaps their jobs. The page you just saw will be the next one you draw on. Which position should you erase there? Look at `previous[page]`. Each page remembers the craft it last displayed.

The two pages each get their own stars, crescent and ridge during setup. BGraph's Primitive_line traces the ridge using the example's `pixel` callback. After that, a frame needs only a 20-by-16-pixel erasure and the small rectangles that make up the new craft. The flight area stays clear of the stationary scenery, so painting the old position black restores the background. Try moving the crescent into the flight area and predict what would happen if you flew through it.

At 320 × 192 with two pixels per byte, one picture uses 30,720 bytes. The library reserves two 32 KiB physical pages, $40000 through $4FFFF. Keep that region free of other data. Use the normal BASIC text screen with interrupts enabled and keep the C program, its data and stack below $8000. Ordinary Hello World settings fit this exercise. Look up the page and memory requirements.

Launch a survey probe

How can one button fire a probe and charge the next one? Use probe.c as **main.c** in a fresh Hello World project. Add **gime**, **input** and **bcontrol** to `LIBS` in that order. Use Disk BASIC with at least 512 KiB of RAM and RGB video. Assign a controller to the **right CoCo port**, or select **Keyboard → Arrows** for that port in **Input → Keyboard and Joysticks…**. In the keyboard setup, **Z** is the primary fire button.

Choose **Build**, then **Run**, and click the CoCo display. Steer right until the middle of your craft is directly above the gold deposit. Center the stick. Press fire and keep it down. One short probe falls and disappears. While you hold the button, watch the gauge at the top left fill with gold. How many probes have you launched?

Release fire when the gauge is full. The gauge turns cyan. Press fire again and keep the craft still. This probe can reach the deposit, which turns cyan when surveyed. Press **Q** after it lands. If you made those two presses, BASIC shows **PROBES LAUNCHED: 2** and **DEPOSIT SURVEYED**.



Figure 25: The survey craft sends a cyan probe toward the gold deposit on the CoCo 3 graphics screen.

Find input_sample in the main loop. Each update takes one snapshot. `stick.pressed & INPUT_PRIMARY` starts a probe and adds one to the launch count. `stick.held & INPUT_PRIMARY` adds energy every update, up to a full

gauge. `stick.released & INPUT_PRIMARY` saves that energy for the next launch. Try releasing early and compare how far your second probe travels.

Give the probe a second landscape

You have a probe, two planets and a question. If you raise a ridge on Mars, will it still be there after a trip home? Each planet gets its own patch of RAM. Your probe uses the same small C array to draw whichever landscape you visit.

Start a fresh **Hello World** project in **C Development**. Save ridge.c and use it as **main.c**. In **Project Settings → Libraries**, add **bank**, **gime** and **hirestxt** to `LIBS`. They supply the terrain storage, the graphics pages and the lettering. Use the **Disk BASIC** profile with at least **512 KiB** and **RGB** video. Choose **Build**, then **Run**. Click the CoCo display to give the probe your keys.

1. Start over Earth. Notice the ridge directly beneath the probe. **A** and **D** move the probe one column at a time. For this experiment, leave it at its starting column, 10.
2. Tap **Space** to visit Mars. Press **R** once. Watch the ridge rise beneath the probe.
3. Tap **Space** to revisit Earth. Did its ridge rise too? Tap **Space** again. The higher ridge is still on Mars. **Q** ends the flight and returns you to BASIC. A new run starts with the original landscapes.

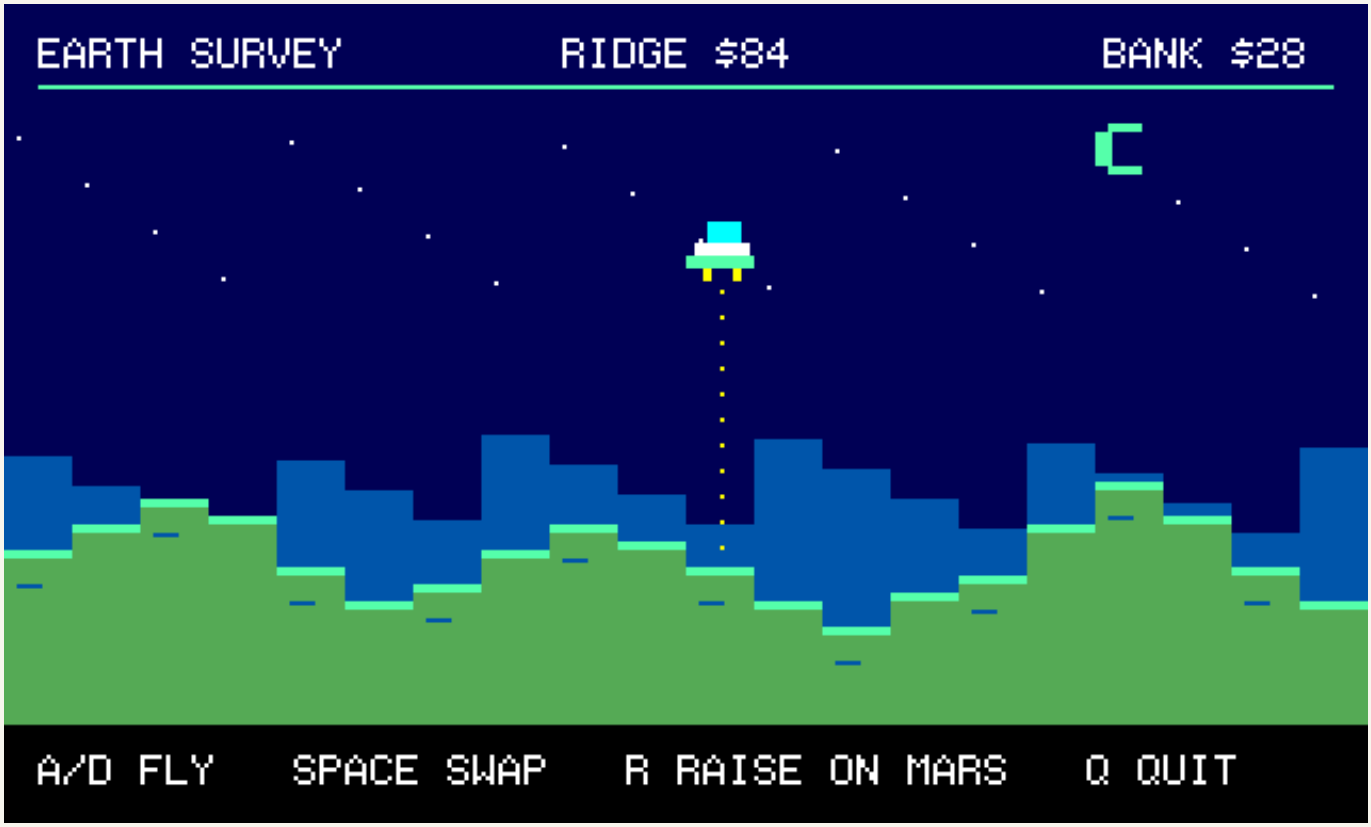


Figure 26: The probe at its starting position over Earth in the running CoCo program.

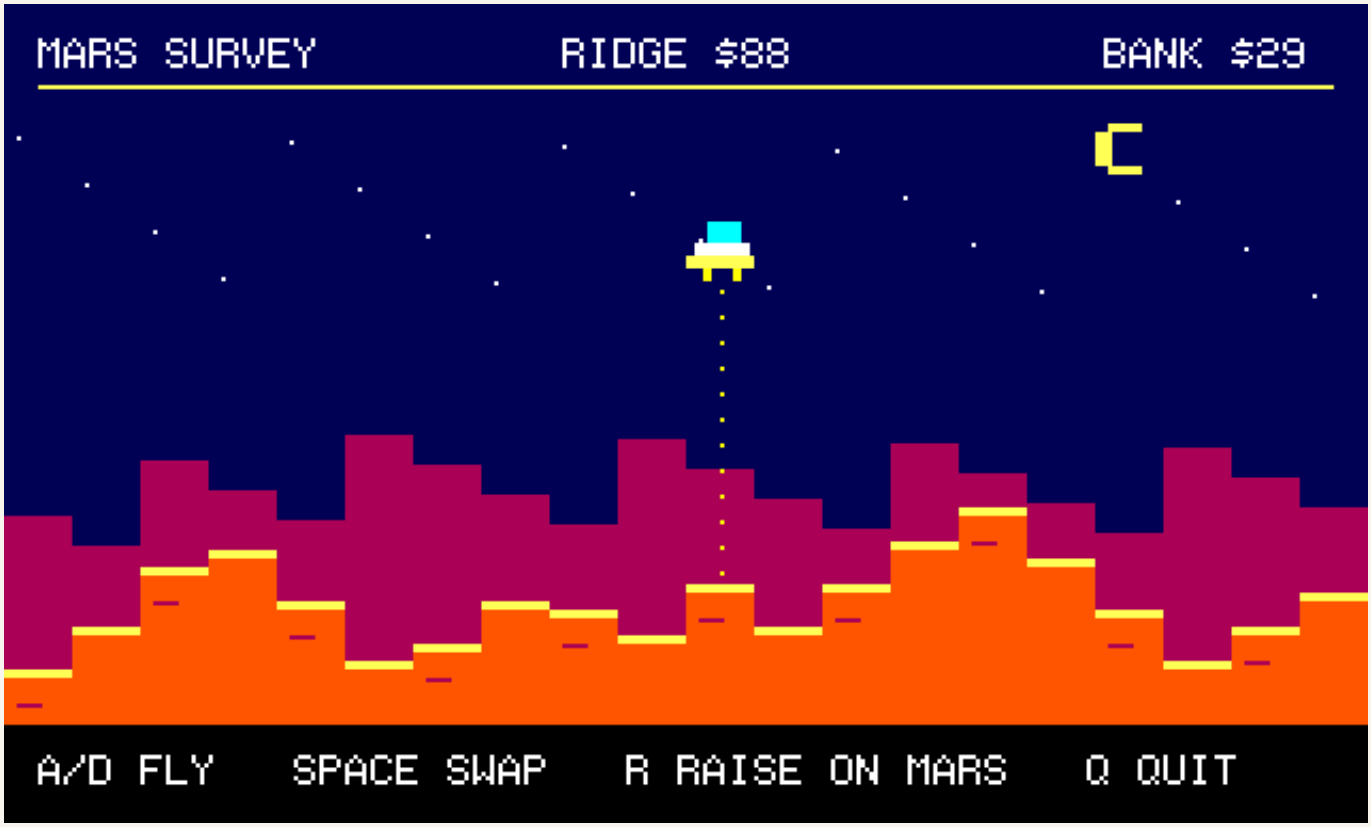


Figure 27: The running program on Mars after raising the ridge beneath the probe and visiting Earth.

Where did Mars go while you were on Earth? A 6809 or 6309 running this program uses a 16-bit address. It can see 64 KiB at any instant, even on a 512 KiB CoCo 3. The memory management unit (MMU) divides that view into eight windows of 8 KiB each. A window can point to a different bank of physical RAM. Here the terrain library briefly borrows the window starting at $8000, copies the requested bytes, then puts the previous mapping back. Your ordinary C array keeps its usual address.

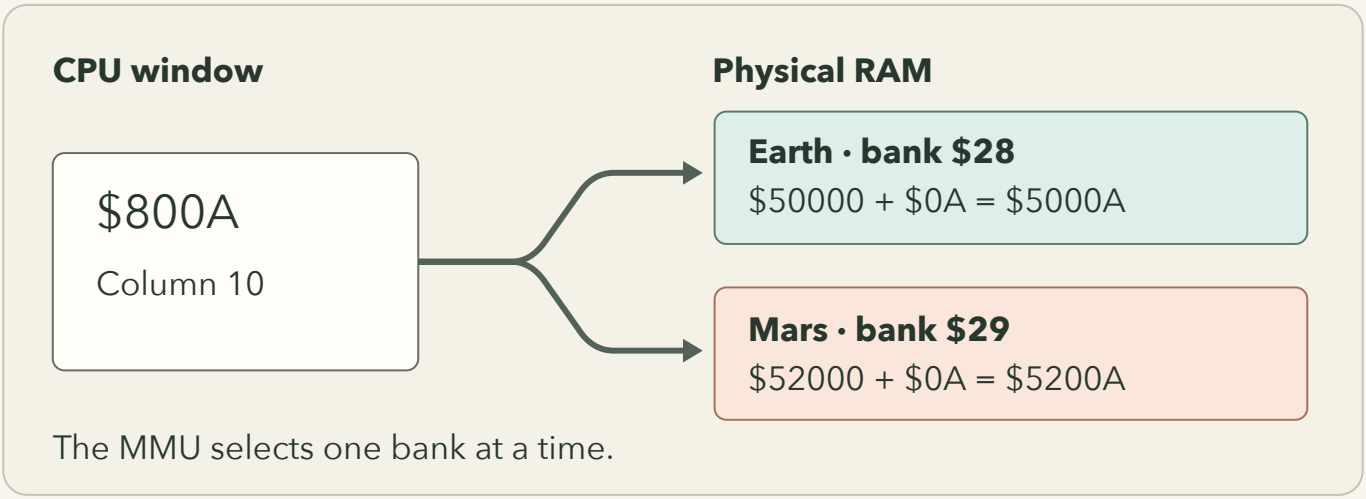


Figure 28: The same CPU address, $800A, reaches Earth at physical $5000A while bank $28 is selected, or Mars at $5200A while bank $29 is selected.

Follow column 10 through the source. Earth starts at offset zero in the reserved storage. Mars starts one bank later, at offset 8,192. Each terrain byte gives the vertical screen coordinate of a ridge top. On Mars the top of the screen reads **RIDGE $98**. That is 152 in decimal. Raising the ridge 16 pixels changes it to **$88**, or 136, because

screen rows count **down**. bank_write saves the edited byte in Mars RAM. The next visit uses bank_read to retrieve the terrain. Try moving over one column before you raise the next ridge. Predict which byte will change.

```
            if (key == 'R' && onMars && ridge[column] >= 96) {
                /* Screen y grows downward. Save just the changed byte. */
                ridge[column] -= 16;
                if (!bank_write(offset + column, &ridge[column], 1))
                    return 0;
                break;
            }
```

Each map uses only 20 bytes. They could easily share one bank. Giving each a bank of its own makes the address change easy to see in the debugger.

See it in the debugger. Start a fresh flight, leave the probe at column 10, switch to Mars and raise the ridge once. Pause the emulator. In the debugger, choose **Memory → Physical RAM**. Enter **5200A** and choose **Go**. Read column **0A** in row **052000**: **88** in hexadecimal is 136. Compare **5000A**, the same column on Earth. It still reads **84**, or 132. The debugger can see both banks while the guest is paused. Resume the emulator to finish your flight. Translate a CPU address in the hardware manual.

Hear a landing signal

Use tone.c with **bsound**. Set the app volume to a comfortable level. The text changes from **BEACON ON** to **BEACON OFF** with a short tone in between.

```
/* SPDX-License-Identifier: GPL-3.0-or-later */
/* Sound a landing signal and give the interrupt vector back. Link bsound. */
#include <Player.h>

int main(void)
{
    Player player;
    printf("BEACON ON\n");
    Player_init(&player, TRUE, NULL);
    Player_sound(130, 12);
    Player_shutdown();
    printf("BEACON OFF\n");
    return 0;
}
```

`Player_init` receives the address of a local Player. Its lifetime covers the whole signal. `TRUE` asks the library to install its own interrupt service. Sixty times per second the running guest signals a tick, allowing the player to finish a note. The first argument of `Player_sound` selects pitch on the CoCo SOUND scale. The second sets duration. Both take values from 1 through 255. This call uses 130 for pitch and 12 for duration, about four-fifths of a second. Keep guest interrupts enabled. `Player_shutdown` releases the interrupt hook before the local Player goes away.

Seeing both messages tells you control returned from the sound call. If both appear in silence, check the emulator's mute and the Mac output volume. Seeing only the first while the emulator is paused calls for the emulator's **Run** button.

Fly the Talking Planet Lander

In C Development, choose **New → Talking Planet Lander**. You can also save the project, expand it, and open the folder containing its Makefile. The program uses the same two destinations as Earth or Mars. This time the craft sends the beacon to your speakers and speaks its destination.

Start the **Disk BASIC** profile. Open **Cartridges**, enable the Multi-Pak, put **Speech/Sound** in slot 1 and **Orchestra-90** in slot 2. Keep the floppy controller in startup slot 4. Speech/Sound needs its PIC and speech firmware. The stereo beacon works with Orchestra-90 in sound-only mode. Use normal guest CPU speed.

Choose **Build** and **Run**. Give the emulator keyboard focus and press **1** or **2**. Look for **DESTINATION EARTH** or **DESTINATION MARS**. Listen for the low note on the left and the higher one on the right. Press either CoCo joystick's primary fire button during an announcement. The screen prints **THRUST** once per press. Both routes finish at **TOUCHDOWN**. A disk you create with **Build Disk** can be mounted in FD0 and started with `LOADM"LANDER":EXEC`.

Open **main.c** and follow three calls:

- snprintf places the destination in a 24-byte array. Its capacity includes the final zero byte. When a future message might outgrow the array, check the return value. A negative result signals an error. A result of 24 or more means only the prefix fit.
- orch90_play sends a block of left/right sample pairs to the stereo converter. It occupies the guest CPU until that block finishes. The two pitches come from how often the sample values alternate. Each quiet channel stays at 128.
- ssc_service feeds one byte of a phrase when the cartridge is ready. Between calls, the program checks the fire buttons. After the final byte, it watches the speech-idle flag for the start and finish of playback, with a timer deadline. You can see why finishing the transfer and finishing the announcement are two different events.

The template already selects **vcccompat**, **orch90**, **ssc** and **bcontrol** under **Project Settings → Libraries**. Use the same selections for another project that needs all four.

Give a planet its own voice

Your crew has named a new planet **Zora**. How do you tell a speech synthesizer what a made-up name sounds like? You can build the name out of *allophones*, the short speech sounds the cartridge can play, and keep a separate English message for the crew.

1. Start the **Disk BASIC** profile. Under **Cartridges**, enable the Multi-Pak and put **Speech/Sound** in slot 1. Supply its PIC and speech firmware if prompted. Keep the floppy controller in startup slot 4. Open **Input → Keyboard and Joysticks** and assign your controller to the right CoCo port. With just a keyboard, choose **Keyboard → Arrows** for that port and use **Z** for fire. Make a note of your original assignment so you can restore it afterward.
2. Save announcer.c as **main.c** in a fresh Hello World project. Select **ssc**, **input** and **bcontrol** under **Project Settings → Libraries**. If you maintain the Makefile yourself, put these three in its library list in that order.
3. Choose **Build**, then **Run**. Click the guest display and press **1** to hear the name. Press the right joystick's primary fire button while the voice plays. Listen for a landing tone that starts strong and fades. Hold the button down. The landing count goes up only once. Release it, let the tone finish, and press again for a second landing.
4. Press **2** for the crew message. Once it finishes, press **2** again. The cartridge can speak the message again without receiving its spelling a second time. Press **Q** when you're ready to return to BASIC.

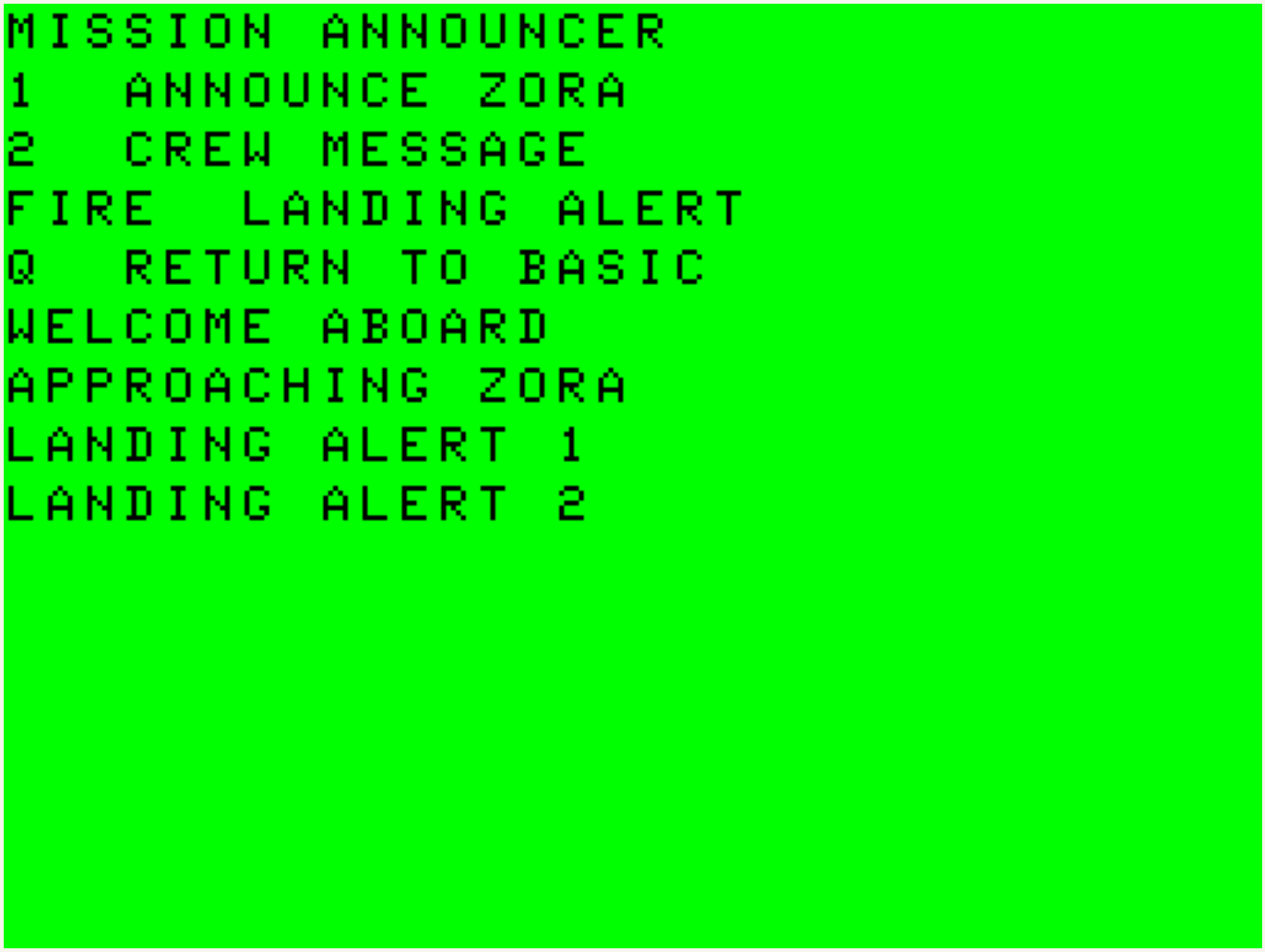


Figure 29: Mission Announcer shows APPROACHING ZORA and two counted landing alerts on the guest screen.

Look at the three calls near the start of the program. ssc_store_allophones puts **ZZ**, **OR** and **AX** in speech slot 1. Say the sounds slowly yourself. The **AX** gives the final syllable its soft "uh." ssc_store_phrase puts the English spelling of **WELCOME ABOARD** in speech slot 3. Both slots belong to the same voice, so let a spoken line finish before asking for another.

ssc_store_enveloped_tone prepares a separate sound slot. **420** sets the pitch period, **SSC_ENV_DECAY** shapes the volume, and **1800** sets how slowly that shape unfolds. The cartridge plays the sound while the main loop reads keys. The program samples the joystick once per guest timer tick because a joystick reading briefly borrows the audio circuit. Try a larger envelope period and listen for a slower fade. The duration of **18** sets when the cartridge silences the channel.

The play calls return **SSC_OK** when the command has been submitted. That is why **APPROACHING ZORA** can appear while you are still hearing the name. The program restores the previous audio route when you quit. If the cartridge times out during a command, it resets the cartridge before returning, which also clears its stored sounds.

For the original cartridge commands and allophone chart, see the Tandy Speech/Sound manual in the bibliography.

Steer while a stereo beacon plays

Imagine flying past a navigation buoy. You hear a low note on your left, a short gap, then a higher note on your right. If your program had to sit in orch90_play for the whole signal, who would steer the ship?

1. Start the **Disk BASIC** profile at its normal guest clock. Under **Cartridges**, enable the Multi-Pak, place **Orchestra-90** in an empty slot and keep the floppy controller in startup slot 4. Sound-only Orchestra-90 works here. In **Input → Keyboard and Joysticks**, assign a controller to the right CoCo port, or choose its keyboard controls and steer with the left and right arrows.
2. In a fresh C **Hello World** project, replace **main.c** with flyby.c. Under **Project Settings → Libraries**, add **orch90**, **input** and **bcontrol**, in that order. Build, run and click the guest display.
3. Find the ship (the up arrow) below the moving star (*****). Press **1** to transmit. Watch the star keep moving as you steer the ship from side to side. Listen for the second note in your right speaker or headphone. Press **Q** mid-note. The signal stops and **FLYBY COMPLETE** appears before BASIC returns.



Figure 30: The Stereo Flyby flight deck with the beacon on air. The pilot steered the ship to the right while the star kept moving.

```
    for (i = 0; i < FRAMES; ++i) {
        /* The first note comes from the left. The right note is an octave up.
           128 in both channels is the quiet space between notes. */
        beacon[i].left = i < 768 ? wave[i & 15] : 128;
        beacon[i].right = i >= 1024 && i < 1792 ? wave[(i << 1) & 15] : 128;
    }
```

Each entry in **beacon** holds one unsigned byte for the left channel and one for the right. The 2,048 frames occupy 4,096 bytes of guest RAM. The middle value, **128**, gives each wave room to swing up or down. Read **makeBeacon** and find which channel sits at 128 during the first note. The right note walks through the same little wave table two places at a time, so you hear it an octave higher. In the running exercise the left speaker

played about 243 cycles per second and the right played about 485 while the pilot steered. Look for the quiet gap after each note. How many of the 2,048 stereo frames does this loop reserve for the two gaps? Count from 768 to just before 1,024, then from 1,792 to the end.

orch90_start leaves the sample in place and returns to the flight loop. Short interrupts feed the next stereo pair to the cartridge. That is why **beacon** lives in a static array. The loop has time to poll the stick and draw a new star position every four BASIC clock ticks. Keep the array mapped and unchanged until orch90_stop returns or a one-shot has finished. End playback before returning to BASIC. The background player needs the standard BASIC interrupt setup. If you adapt another exercise that uses a BSound player, shut that player down before you start the beacon.

Read the Descent Computer

Your lander's radar reports 105.50 metres. Two seconds later it reports 101.75 metres. How fast are you falling? The pilot wants a short, readable instrument panel. The computer needs the full speed for its next calculation. Start with the **Disk BASIC** profile and a fresh C **Hello World** project. Save descent.c as **main.c** and add **vcccompat** under **Project Settings → Libraries**. Use the default CoCo BASIC floating-point backend. Build and run. The program reads its two simulated radar heights, prints one panel and returns to BASIC.

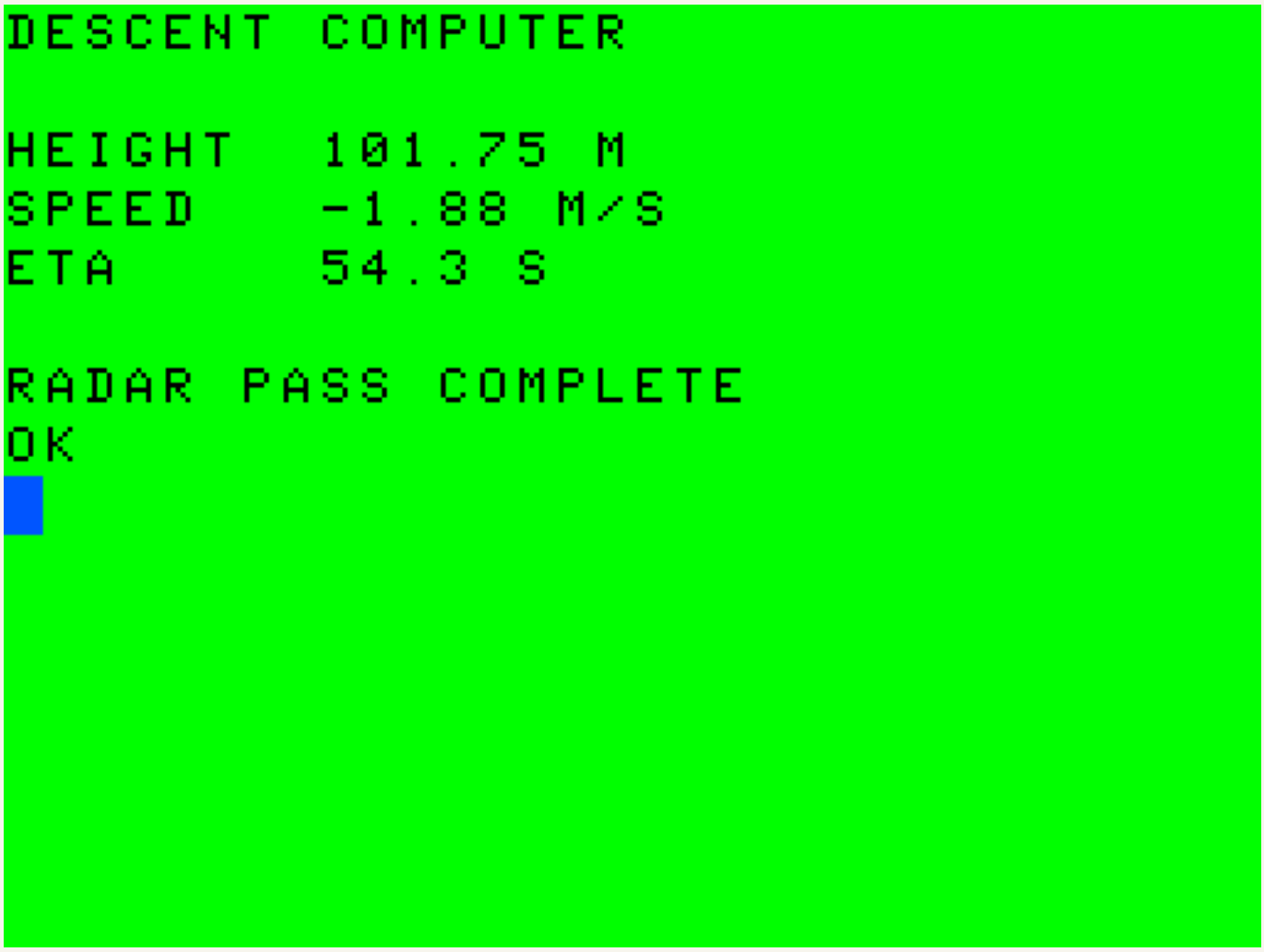


Figure 31: The Descent Computer guest display reads HEIGHT 101.75 M, SPEED −1.88 M/S and ETA 54.3 S.

```
    /* Falling gives a negative speed. Keep the full value for the estimate. */
    speed = (lastHeight - firstHeight) / seconds;
    reading("HEIGHT  ", lastHeight, 2, "M");
    reading("SPEED   ", speed, 2, "M/S");
    if (speed < 0.0f)
        reading("ETA     ", lastHeight / -speed, 1, "S");
    else
        printf("ETA     NO DESCENT\n");
```

Subtract the earlier height from the later one. The change is −3.75 metres in two seconds, so the vertical speed is −1.875 metres per second. The minus sign means the lander is getting closer to the surface. The panel rounds that reading to **−1.88**. The landing estimate still uses the original speed. At that steady rate, covering the remaining 101.75 metres would take about **54.3 seconds**. If the two heights are equal, the craft is hovering. The program then prints **NO DESCENT** in the ETA field and skips the division.

```
    char field[12];
    int length = vcc_fixed(field, sizeof(field), value, places);
    if (length < 0 || (unsigned)length >= sizeof(field))
        printf("%s--- %s\n", label, unit);
    else
        printf("%s%s %s\n", label, field, unit);
```

vcc_fixed writes the number into **field**. Its return value tells you how many characters the whole number needs, excluding the final zero byte. The text **101.75** needs six characters and a seventh byte to mark the end. If the whole number will not fit, the instrument prints three dashes. A truncated speed could give a pilot the wrong idea.

Try making **field** five bytes long. Predict which of the three readings survives. The rounded speed **−1.88** takes five characters *before* its terminator. The ETA **54.3** takes four. Restore the original size when you finish.

Make room for one more crew member

Your ship has two empty seats and expects a third passenger. How can a program grow its seat list on a CoCo? Save crew.c as the **main.c** of a fresh Hello World project. Under **Project Settings** → **Libraries**, add **vcccompat**, then **wydealloc**. For the Hello World Makefile, append both to `LIBS`. Choose **Build**, then **Run**.

Find `storage[96]`. Those 96 bytes are the entire heap budget, including the space the allocator uses to keep track of each block. The array is static, so its address stays valid for the life of the program. The call to vcc_heap_init hands this region to the allocator. Leave the array alone once you've handed it over. The compiler's stack setting still needs room for local variables and function calls.

calloc supplies two zeroed seats. After the program fills them, realloc asks for room for a third. Notice the second pointer, **larger**. The program assigns it to **seats** only after it knows the request succeeded. It then writes the new seat before reading it.

The guest prints **CREW AT LAUNCH: 0** and **CREW AFTER GROWTH: 3**. Predict the next two lines. The program asks for 200 bytes. The entire heap has only 96! The guest says **NO ROOM FOR 200**, then **CREW STILL ABOARD: 3**. The original three entries survived the failed request. free gives their space back, and **CREW LOG CLOSED** appears before BASIC returns.

For a list that grows with outside input, check every allocation this way. Even when the total free bytes look sufficient, one request needs a single contiguous block. During growth the old and new blocks may need to fit at the same time. The pool.c exercise uses Pierre Sarrazin's WydeAlloc directly if you want to manage several separate pools.

Five more experiments

Use the same private-project steps for any of these. Each example needs just the library in its row. The files contain the initialization and cleanup as well as the call doing the interesting work.

Try this file	Library and header	What to look for
pool.c	**wydealloc**, `wydealloc.h`	Our own 64-byte storage pool has room for an eight-byte block. An 80-byte request returns a null pointer. The screen says **SMALL OK, LARGE REFUSED** and **AFTER FREE: OK**. Follow which pointer we release before asking for eight bytes again.
glyph.c	**hirestxt**, `hirestxt.h`	Read **MARS STATION** on a graphics screen. Press a key to return to hardware text. The letters are pixels written into the PMODE 4 buffer. Use a fresh Disk BASIC setup with its standard four graphics pages reserved and the normal project load address.
route.c	**astar**, `astar.h`	The rooms sit in a row. The last room is the goal. After `astar_searchGrid` finds it, the iterator reports **BACK FROM GOAL: 2 1 0**. Reverse that order if you want directions for the outward journey.
deal.c	**cardgame**, `cardgame.h`	Three numbered cards start in a known order. `pickCardFromDeck` removes the card at the end and updates the count through its address. Expect **DREW 3, 2 LEFT**. The small example never needs a card bitmap.
editlog.c	**ebitor**, `Ebitor.h` and `sg4term.h`	Two simulated keys add O and K to a 32-byte edit buffer. Moving the cursor left commits the inserted run to that buffer. The final report is **LOG: OK**. Compare the guest's editing buffer with the C source open on the Mac.



Figure 32: MARS STATION appears as compact software-drawn letters at the upper left of a real guest graphics screen.

A header tells you what you may call. The host library settings tell the build where to find the implementation. The machine still has a say in what you see. Check the screen mode, the controller assignment or the sound level before changing a working algorithm.

Fly the Mission Recorder

Your survey ship flies over sixteen sectors. Mark one from the cockpit, fly on and file a report. Which sector will Mission Control receive after you fly away from your mark?

1. Start **Disk BASIC** with **512 KiB** of RAM, normal guest speed and **RGB** video. Keep the **FD-502** in Multi-Pak startup slot **4**. Mount an initialized, writable practice disk in **FD0**. You can save a blank practice disk from the offline library. Open **Media → Printer Output…** and choose **Start Capture** to keep the printed record.
2. Save the complete Mission Recorder project. Expand the ZIP in Finder. Under **Develop → C Development…**, choose **Open**, select the **MissionRecorder** folder containing the Makefile, then choose **Build** and **Run**. The workbench loads the binary into the CoCo. Your practice disk stays in FD0 to receive the report.
3. Click the guest display. Choose **1** for a silent flight. Watch for the ship above the ridge in sector **07**. When it appears, both screen pages are ready and you can steer. Tap **D** to move right until the readout reaches **15**, or steer there with the **right CoCo joystick**. Press **Space** or the stick's primary fire button to mark the

sector. A gold marker appears. Move left with **A** or the stick. Watch **SECTOR** follow the ship while **MARK** stays at **15**. Press **S** to finish and save. **Q** instead discards the flight, including an unsaved mark.

4. The first report on the blank practice disk is **LOG01.TXT**. Look in **Printer Output** for two lines, **MISSION RECORDER** and **SECTOR 15**. The ship moved after you marked the sector, yet the printer still reports **15**. Back at BASIC, type **DIR 0** to find the disk file. Choose **Run** in the workbench to fly again. A second save on the same disk gets a new filename.

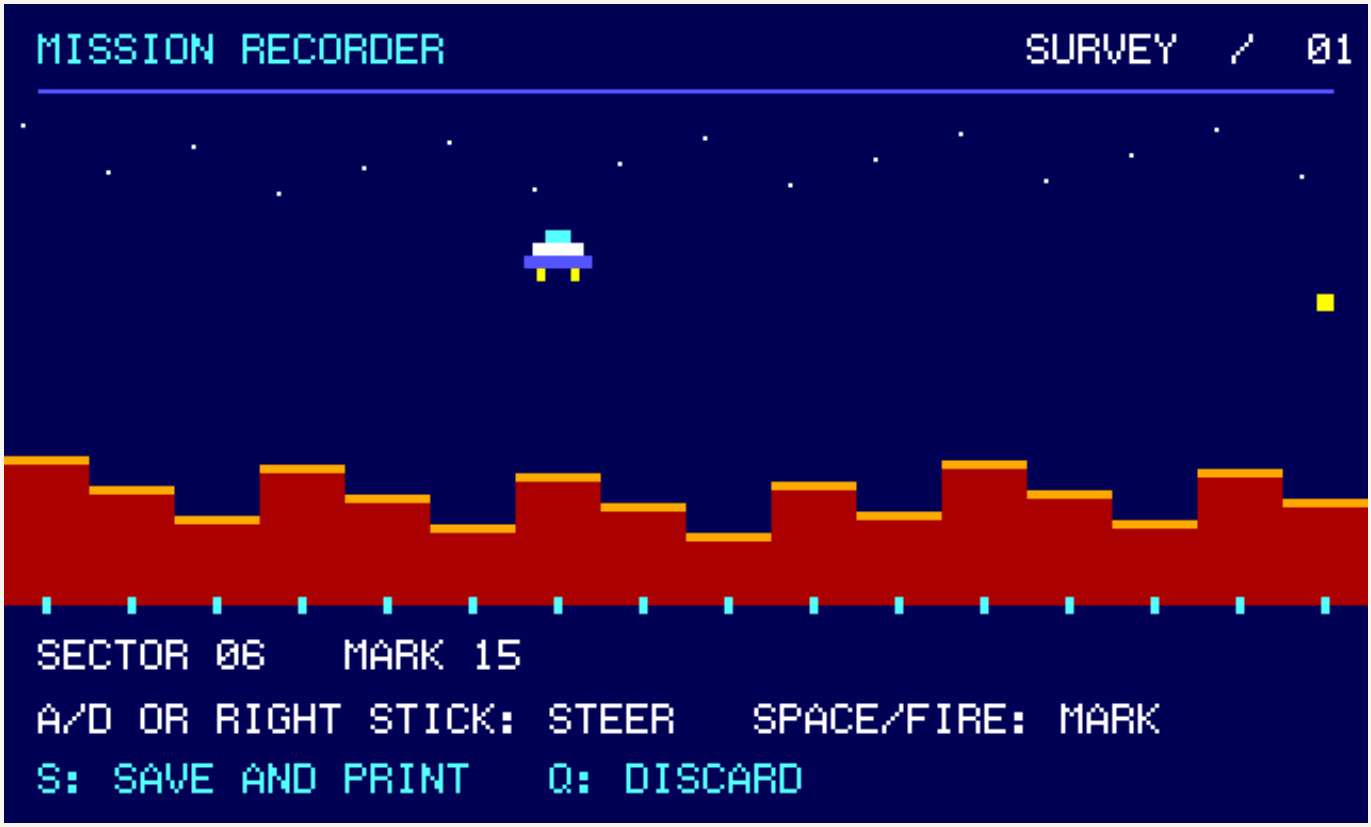


Figure 33: The flight display reads SECTOR 06 and MARK 15. The gold marker remains at the far right, where the pilot recorded it.

Take the program on disk. Choose **Build Disk** and **Show Build Products**. Copy **RECORDER.dsk** to your own folder, mount the copy in FD0 and type `LOADM"RECORDER":EXEC` in BASIC. The copy holds your reports beside the program and survives a rebuild. To keep a separate copy of the printed lines on the Mac, use **Save Text...** in Printer Output.

Follow the observation through the source. Open **flight.c** to find the mark, **report.c** to find the disk write, and **main.c** to see when each runs. The pilot's first mark stores a sector number in banked RAM. Steering and redrawing the cockpit can now continue without changing that observation. The save routine writes a temporary file, closes it and reads it back. If every byte matches and the file ends there, it renames the draft from **LOG01.TMP** to **LOG01.TXT** and sends the same two lines to the printer. If either name already exists, the recorder tries the next number, up to 99. An older report or a draft left by an interrupted save therefore keeps its name and contents. If the disk is write-protected, leave the guest at the retry prompt, remove write protection from the practice disk and press **R**. The original mark survives the retry. The printer receives nothing from a failed save.

Add sound for a second flight. In **Cartridges**, place **Speech/Sound** in Multi-Pak slot **1** and **Orchestra-90** in slot **2**. Supply the two Speech/Sound firmware files if prompted. Orchestra-90 can run in sound-only mode. Pick **2** for speech, **3** for stereo or **4** for both. The choice tells your program which cartridges are available. With just the disk controller, choose **1**. On a stereo flight, the left and right channels take turns sounding a beacon. On a speech flight, marking the sector announces "mark recorded." The flight loop offers the cartridge one byte at a time, so it can still respond to your stick while the announcement is being sent.

Find **beacon** at the top of **flight.c**. Its 32 stereo frames occupy 64 bytes, one byte per channel per frame. The foreground loop steers the ship and draws the screen while the Orchestra-90 player borrows BASIC's interrupt handler to feed the next frame. Keep that array in ordinary mapped memory and leave interrupts enabled during flight. Before disk or printer I/O, the program stops the player, restores the speech route if it used it, and returns BASIC's palette and memory mapping. Stereo Flyby isolates the beacon technique.

Know which RAM belongs to whom. The graphics library reserves **64 KiB** for the two screen pages in physical banks **$20–$27**. The bank library reserves the next **32 KiB** (**$28–$2B**). This flight writes just one byte at offset zero of that second area. The survey paints the ridge once on each screen page. Each page remembers where its own ship and marker were drawn. On the next turn the pilot only has to redraw the changed parts. Neither screen can overwrite the mark in its separate bank. The banked terrain exercise shows how the processor can reach that byte even while the display controller reads a different page. After a restart you can read the report from your practice disk.

How much room to add an instrument? The link map counts code and static data, including space for initially empty buffers. In the supplied project the Debug build occupies **22,513 bytes** from **$2000** up to **$77F1** (the first unused address). Release occupies **21,775 bytes**, up to **$750F**. That leaves just **15** and **753** bytes respectively before the project's **$7800** link limit. The project budgets a separate **2 KiB** for the stack. Even adding a message can push Debug over its link limit. If that happens, try Release or remove an optional feature before changing the memory layout. Keep a large terrain table in the reserved banked area and copy a small piece at a time. Rebuild and look at **RECORDER.map** to measure your own version.

Hand someone a disk they can run

Chapter 9 of 9

Two people want your project. A friend wants to try it. Another programmer wants to add a seventh word. Prepare two different handoffs and test what each person gets.

Pack a six-word disk

1. Start a fresh **Hello Disk** project in its own folder. In **assets/words.txt** add **ORBIT** on a new line after PLUS. Use uppercase letters. The supplied generator skips blank lines, accepts one A–Z word per nonblank line and requires the encoded list, including its returns and terminator, to fit in 256 bytes.
2. Save, then **Build Disk**. Open **Show Build Products** and find **HELLO.dsk**. Copy it to a separate folder and give the copy a recognizable Mac filename such as **Orbit Crew.dsk**. The guest executable inside still has the name HELLO.
3. Finish any pending writes and preserve open guest work. Start at **OK** on a fresh **Disk BASIC** machine. In the floppy controls, mount your copied image in **FD0**. This test starts from the disk you will share, independently of the C editor.
4. Type **DOS** and press Return. Look for **ORBIT** beneath **PLUS**, followed by **DISK RESULT: 0** and **GENERATED WORD COUNT: 6**.
5. On a fresh start with that copy mounted, type **LOADM"HELLO":EXEC** and press Return. Look for the same three clues. This route loads the executable by name and transfers control to it.

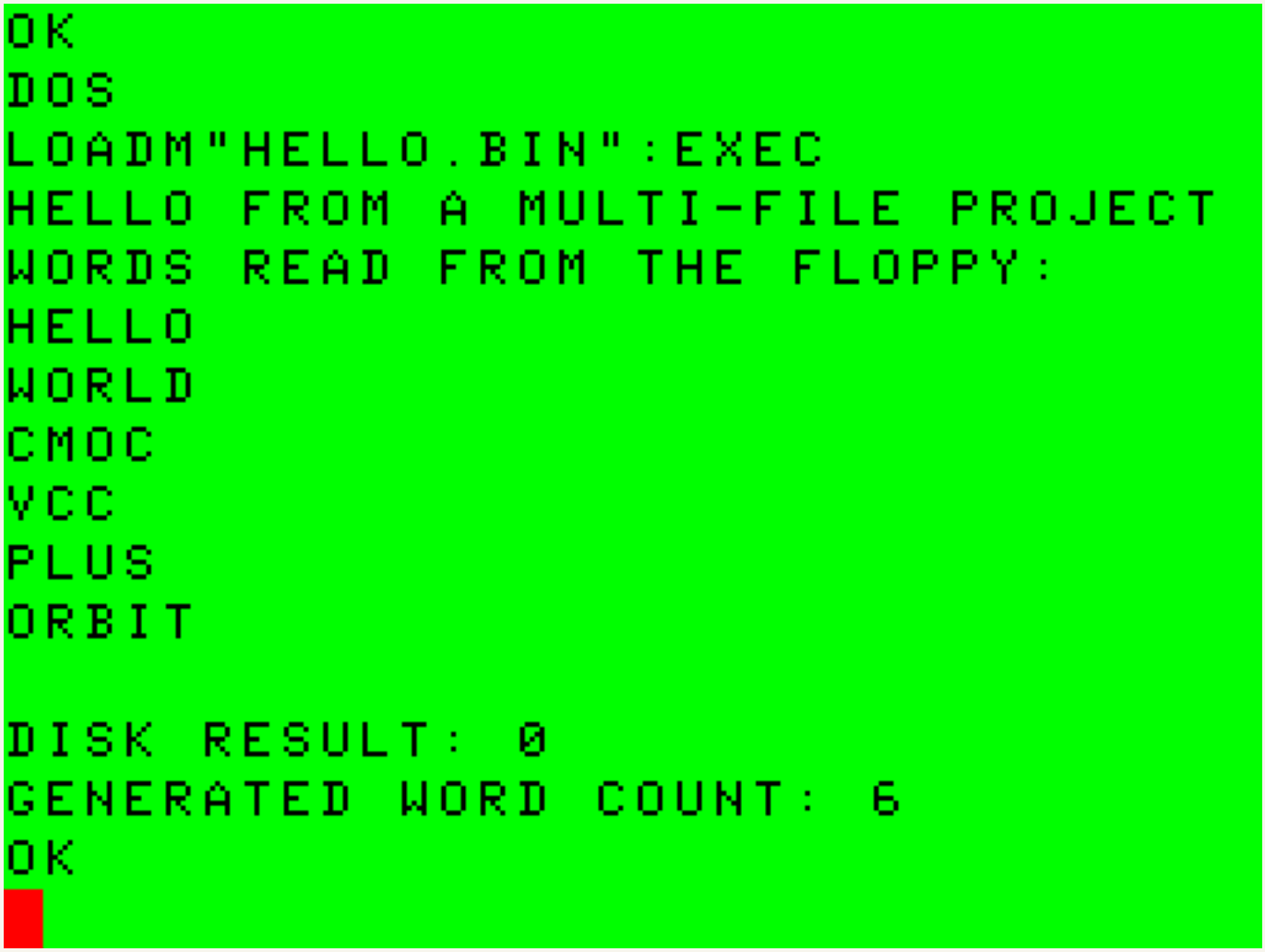


Figure 34: A fresh Disk BASIC guest ran the copied disk with DOS. ORBIT appears after PLUS, and the generated count is now 6.

Write those two commands on the disk label you send your friend. Typing DOS works because the generated project installs a boot helper on track 34. **Project Settings → Disk & Run → DOS boot → Run this program** controls that choice. The helper feeds Disk BASIC the load-and-execute command. It consumes one track of the generated floppy. The disk and the instructions are enough for a player with a compatible Disk BASIC setup.

Hand over the recipe too

For the programmer, copy the saved **project folder**: the Makefile, project configuration, source, headers, word list and generator. A copy of HELLO.dsk is useful for comparison. It cannot tell the new programmer where you chose the word ORBIT.

Close the project, then use **Open** in C Development to select the copied folder. Check its word list, select **Debug**, and **Build Disk**. Launch the result. Read the sixth word and count. This last test catches an easy packaging mistake: copying an older source folder alongside a newer floppy.

Open **Build History** to identify the build you just made. Check the configuration, build action and the Program or Disk indicators before using its Finder button. If you return to the project later, changed sources can make an earlier executable unsuitable for debugging. Keep the source and the tested image in your handoff folder even if you later clean out older build products.

Bring someone else's Makefile

A downloaded project may call its default target **all**, a program filename, or something else. Open the trusted source folder and look at its Makefile before building. In **Project Settings** → **Build**, choose **Custom Makefile** to keep its rules. A blank ordinary build target lets Make choose its default. Set **Build Disk**, the expected binary and disk names, and **Disk & Run** from the rules you actually find. A successful target that builds only a binary cannot supply a missing data disk.

In **Libraries**, a custom project can bind a Makefile *directory variable* to a bundled library, or append the library to a *linker variable* used by its link recipe. Enter the name used by that Makefile and choose a minimum version when one matters. A recipe that rebuilds a local library from its own sources needs those sources. A prepared archive satisfies a recipe that links an existing library. Check **Preview** for generated projects and the build output for either kind.

If you also hand the project to an assistant, open it in C Development and export a connection for that project. Custom recipes and explicitly shared support folders need local approval; the connection does not grant access to an entire checkout by implication. Keep the original Makefile in control of its build, and retain the matching debug evidence if the next developer needs verified source stops. A disposable test disk is a working result, while the source folder and recipe remain the reproducible handoff.

To see why a guest program can depend on a ROM even when every Mac tool is bundled, return to the disk reader. For an unfamiliar include or linker name, open Find a call.

Find a call

Appendix A

A disk file starts at sector zero. A physical floppy starts numbering its sectors at one. Which number belongs in a C call? Pick the **job** below, or choose the header already at the top of your source file. Each symbol has its own bookmark. The book's Search also accepts an exact spelling such as `decb_readSector`.

For declarations, `byte` means an eight-bit unsigned value and `word` a 16-bit unsigned value. An asterisk identifies an address in the guest. When an entry asks for a writable buffer, the caller owns the storage and keeps it alive until the operation finishes.

Start with a job

- Browse the SDC card · sdc_list_begin
- Switch a data disk · sdc_mount
- Read a line from a guest disk file · fgets
- Split a record, then split one of its fields · strtok_r
- Read records or exact binary bytes · fread
- Write a numbered observation directly to a file · fprintf
- Write a record and make it readable on disk · fflush
- Retry a refused save on the same disk · clearerr
- Keep a backup before installing a replacement · rename
- Delete a closed file from a guest disk · remove
- Read a disk sector from an open file · decb_readSector
- Read a physical track and sector · readDiskSector
- Open a named floppy file · decb_openSectorFile
- Report a missing file · DECB_ERR_NOT_FOUND
- Exchange a cassette file with BASIC · vcc_tape_read_line
- Send a report to the TP-10 · vcc_printer_printf
- Show a radar reading with fixed decimal places · vcc_fixed
- Print a greeting · printf
- Check a number typed at the keyboard · strtoul
- Read a signed decimal or hexadecimal value · strtol
- Measure a string · strlen
- Change text width · width
- Color a text cell · attr
- Fit a name in a bounded buffer · memcpy
- Poll a key · inkey
- Plot a 16-color pixel · Primitive_setPixel320x16
- Copy terrain from banked RAM · bank_read
- Show a completed graphics frame · gime_present
- Fire once on a joystick press · input_sample
- Read a joystick · Joystick_getAxisValues
- Play a tone · Player_sound
- Give an invented name its own pronunciation · ssc_store_allophones
- Build a fading landing alert · ssc_store_enveloped_tone
- Speak a phrase · ssc_say
- Send stereo samples · orch90_play
- Keep stereo playing while a game runs · orch90_start
- Format text safely into a buffer · snprintf
- Print in a graphics buffer · initHiResTextScreen2
- Reserve a bounded guest heap · vcc_heap_init
- Grow an allocation · realloc
- Allocate from a pool · MemPool_alloc
- Trace a grid path · astar_searchGrid

- Deal a card · pickCardFromDeck
- Edit a text buffer · Ebitor_processKey

Start with a header

Choose the feature's header to see its public names and build setting. Two libraries can choose the same spelling. For example HiresTxt and CardGame each supply a moveCursor routine for their own renderer.

- vcc/decimal.h · Format a CoCo BASIC floating-point number into a bounded field.
- vcc/cassette.h · Write and read named text files on cassette.
- vcc/printer.h · Send formatted text to the serial printer.
- vcc/sdc.h · Browse the SDC card and choose the image in either drive.
- vcc/bank.h · Keep assets in four reserved CoCo 3 memory banks.
- vcc/input.h · Sample a joystick and recognize presses, holds and releases.
- vcc/gime.h · Draw on a hidden CoCo 3 graphics page and display the completed frame.
- vcc/ssc.h · Speak a phrase or play a tone through the Speech/Sound Cartridge.
- vcc/orch90.h · Play stereo sample frames through Orchestra-90.
- ctype.h · Classify ASCII bytes and convert letter case.
- errno.h · Check whether a number was outside the conversion range.
- limits.h · Look up the ranges of CMOC integer types.
- stddef.h · Use CMOC sizes, member offsets and pointer differences.
- stdint.h · Use explicit 8-, 16- and 32-bit types and limits.
- stdio.h · Read a guest disk file or format text into a bounded buffer.
- stdlib.h · Convert a number from text or manage a bounded guest heap.
- string.h · Split a string into tokens, compare text or move bytes.
- 8-bit-wide-font.h · Find the game renderer's eight-pixel-wide glyph data.
- assert.h · Check program assumptions at build time and while running.
- astar.h · Find a path across a rectangular grid and walk back from the goal.
- cardgame.h · Build a deck, select a card and render game graphics.
- catmull-rom.h · Interpolate a smooth path through a sequence of points.
- clipping.h · Clip a line to a rectangle before drawing it.
- cmoc.h · Console output, strings, memory, arithmetic and callbacks.
- coco.h · BASIC display modes, keys, joysticks, sound and timing.
- compiled-cards.h · Find the precomputed suit symbols, ranks and edges used for drawing cards.
- decbfile.h · Read and write named Disk BASIC files, sectors and fixed-size records.
- disk.h · Read a Disk BASIC file or address a physical floppy sector.
- Drawer.h · Move a drawing pen and run a DRAW script in a graphics buffer.
- dskcon-standalone.h · Operate the floppy controller while the Disk BASIC ROM is banked out.
- Ebitor.h · Maintain an editable text buffer and a cursor through terminal callbacks.
- font4x8.h · Find the narrow four-pixel glyph data for HiresTxt.
- font5x8.h · Find the five-pixel glyph data for HiresTxt.
- gimeterm.h · Implement terminal operations using the CoCo 3 GIME hardware text screen.
- hirestxt.h · Render text and a cursor into a graphics screen.
- Joystick.h · Sample analog axes and detect joystick button presses.
- Keyboard.h · Poll a guest keyboard with CoCo 3 control, function and navigation keys.
- lores.h · Read or change the color of a low-resolution semigraphics block.
- Painter.h · Flood-fill a connected area with caller-owned scratch space.
- Player.h · Generate music with the CoCo sound DAC and interrupt timer.
- pmode4game.h · Set up the game buffer, graphics cursor, text and keyboard interaction.
- Primitive.h · Plot one pixel or draw lines and shapes using a buffer-specific callback.
- semigr.h · Manipulate legacy semigraphics pixel buffers.
- setjmp.h · Save and restore a nonlocal return point within a still-active call.
- sg4term.h · Implement terminal operations on a traditional CoCo character screen.
- SquareWaveGenerator.h · Produce square-wave samples for a chosen amplitude and period.
- stdarg.h · Pass a variable-length argument list through your own function.
- Trig.h · Use 15-degree sine and cosine increments without floating point.
- Window.h · Describe the guest text window used by the Ebitor terminal adapters.

- wydealloc.h · Manage variable-size allocations inside a buffer your program owns.

Which edition do I have?

This development edition indexes CMOC 0.1.101 targeting CoCo BASIC. Its C declarations were checked against the reconciled runtime headers and their target archive exports. The matching compiler and linker also resolved the addresses of the listed functions and data. This is a development-runtime audit, not a claim that the published application has changed. A **See it used** link identifies a worked program containing that symbol. Other entries document its interface. The hardware experiments and their observed results are in Choose a library for a job.

The standard CoCo floating-point backend calls BASIC. The optional MC6839 ROM backend and its conditional log2f and setMC6839Trap entries require a separately supplied archive. It was absent from the reviewed installation. The separately staged standalone disk driver **was** present. A public build without that option can use ROM-backed decbfile. DriveWire helpers, the file chooser and auxiliary binary loaders in the upstream disk library fall outside this installed subset.

The OS-9 terminal/timer headers and Vectrex headers belong to other targets. Headers whose names include *private*, the internal assertion implementation and the compiler's internal variable-argument helper serve library implementations. cc3graph.h appears in BGraph sources but is outside this installed public header set.

For compiler, library and optional-driver credits, open **Licenses & Notices** in VCC Plus. The source shelf identifies the editions these entries describe.

cmoc.h

Console output, strings, memory, arithmetic and callbacks. The CoCo target supplies this header and runtime automatically.

abs · adddww · atoff · atoi · atol · atol16 · atoui · atoul · atoul16 · bsearch · cmpdww · ConsoleOutHook · delay · div328 · divdwb · divdww · divmod16 · divmod8 · dwtoa · enable_printf_float · exit · ftoa · isalnum · isalpha · isdigit · isspace · itoa · itoa10 · labs · logf · ltoa · ltoa10 · memchr · memcmp · memcpy · memichr · memicmp · memmove · memset · memset16 · mulwb · mulww · NULL · offsetof · printf · putchar · putstr · qsort · rand · RAND_MAX · readline · readword · sbrk · sbrkmax · set_null_ptr_handler · set_stack_overflow_handler · setConsoleOutHook · SIZE_MAX · size_t · sprintf · sqrt16 · sqrt32 · srand · SSIZE_MAX · ssize_t · strcat · strchr · strcmp · strcpy · strcspn · stricmp · strlen · strlwr · strncat · strncmp · strncpy · strpbrk · strrchr · strspn · strstr · strtof · strtok · strtol · strtol10 · strtol16 · strtoul · strtoul10 · strtoul16 · strupr · subdww · tolower · toupper · ultoa · ultoa10 · ultoa16 · utoa · utoa10 · vprintf · vsprintf · zerodw

abs

```
int abs(int j);
```

Return the magnitude as a signed value of the same width. The most negative input has no positive counterpart in that representation.

cmoc.h

adddww

```
void adddww(unsigned *twoWords, unsigned term);
```

Treat two words, high word first, as one unsigned 32-bit quantity. The first three calls update it in place. Comparison returns −1, 0 or +1.

cmoc.h

atoff

```
float atoff(const char *nptr);
```

Parse a decimal floating-point number. In the BASIC backend use uppercase E for an exponent; spaces can occur within a number. strtof can report the first unconsumed character through endptr.

cmoc.h

atoi

```
int atoi(const char *s);
```

Read decimal digits from a string. Match the returned signedness and width to your data; these calls have no error-output parameter.

cmoc.h

atol

```
long atol(const char *nptr);
```

Read decimal digits from a string. Match the returned signedness and width to your data; these calls have no error-output parameter.

cmoc.h

atol16

```
long atol16(const char *nptr);
```

Read hexadecimal digits from a string into a 32-bit result. Choose the signed or unsigned form appropriate to the caller.

cmoc.h

atoui

```
unsigned atoui(const char *s);
```

Read decimal digits from a string. Match the returned signedness and width to your data; these calls have no error-output parameter.

cmoc.h

atoul

```
unsigned long atoul(const char *nptr);
```

Read decimal digits from a string. Match the returned signedness and width to your data; these calls have no error-output parameter.

cmoc.h

atoul16

```
unsigned long atoul16(const char *nptr);
```

Read hexadecimal digits from a string into a 32-bit result. Choose the signed or unsigned form appropriate to the caller.

cmoc.h

bsearch

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *key, const void *element));
```

Supply the size of one element in bytes and a consistent three-way comparator. qsort reorders the array in place. bsearch expects that ordering and returns a pointer into the array or NULL.

cmoc.h

cmpdww

```
signed char cmpdww(unsigned left[2], unsigned right);
```

Treat two words, high word first, as one unsigned 32-bit quantity. The first three calls update it in place. Comparison returns −1, 0 or +1.

cmoc.h

ConsoleOutHook

```
typedef void (*ConsoleOutHook)(void);
```

Install a character-output routine and keep the previous hook to restore later. The character arrives in register A. Preserve B, X, Y and U, including when the program calls printf.

cmoc.h

delay

```
void delay(size_t sixtiethsOfASecond);
```

Wait for the specified number of 60 Hz ticks. A value of 60 is about a second at the normal guest timing.

cmoc.h

div328

```
div328 = divdwb
```

Divide the unsigned value in two consecutive words by a nonzero divisor. The quotient replaces those words. div328 is the former spelling of divdwb.

cmoc.h

divdwb

```
void divdwb(unsigned dividendInQuotientOut[2], unsigned char divisor);
```

Divide the unsigned value in two consecutive words by a nonzero divisor. The quotient replaces those words. div328 is the former spelling of divdwb.

cmoc.h

divdww

```
void divdww(unsigned dividendInQuotientOut[2], unsigned divisor);
```

Divide the unsigned value in two consecutive words by a nonzero divisor. The quotient replaces those words. div328 is the former spelling of divdwb.

cmoc.h

divmod16

```
void divmod16(unsigned dividend, unsigned divisor, unsigned *quotient, unsigned *remainder);
```

Divide by a nonzero value. Supply storage for both outputs; quotient and remainder use the width in the declaration.

cmoc.h

divmod8

```
void divmod8(unsigned char dividend, unsigned char divisor, unsigned char *quotient, unsigned char
*remainder);
```

Divide by a nonzero value. Supply storage for both outputs; quotient and remainder use the width in the declaration.

cmoc.h

dwtoa

```
char *dwtoa(char *out, unsigned hi, unsigned lo);
```

Treat hi and lo as halves of one unsigned 32-bit value. Supply 11 bytes. Print from the returned pointer, which skips leading zeros.

cmoc.h

enable_printf_float

```
void enable_printf_float(void);
```

Enables %f when using the separately supplied MC6839 backend. The standard CoCo BASIC float configuration uses its own initialization.
cmoc.h

exit

```
void exit(int status);
```

End the current program. A return into BASIC requires its memory and execution environment to remain usable.
cmoc.h

ftoa

```
char *ftoa(char out[38], float f);
```

Give the conversion a writable array of at least 38 bytes. The return is the start of the string on success or NULL on failure.
cmoc.h

isalnum

```
int isalnum(int c);
```

Test a character for the property named in the function. A nonzero result means it belongs to the class.
cmoc.h

isalpha

```
int isalpha(int c);
```

Test a character for the property named in the function. A nonzero result means it belongs to the class.
cmoc.h

isdigit

```
int isdigit(int c);
```

Test a character for the property named in the function. A nonzero result means it belongs to the class.
cmoc.h

isspace

```
int isspace(int c);
```

Test a character for the property named in the function. A nonzero result means it belongs to the class.
cmoc.h

itoa

```
itoa(value,str,base)
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

itoa10

```
char *itoa10(int value, char *str);
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

labs

```
long int labs(long int j);
```

Return the magnitude as a signed value of the same width. The most negative input has no positive counterpart in that representation.
cmoc.h

logf

```
float logf(float x);
```

These functions use the selected floating-point backend. For the BASIC backend keep its ROM mapped. logf requires a positive argument and sqrtf requires zero or greater; a BASIC math error can end the program.
cmoc.h

ltoa

```
ltoa(value,str,base)
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

ltoa10

```
char *ltoa10(long value, char *str);
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

memchr

```
void *memchr(const void *s, int c, size_t n);
```

Search the first n bytes for c. The result points inside the original range, or is NULL. memichr folds letter case.
cmoc.h

memcmp

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Compare exactly n bytes of two readable ranges, including zero bytes. Zero means a match. memicmp ignores ASCII case.
cmoc.h

memcpy

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copy n bytes into caller-owned storage and return its address. Size both regions for n. Choose memmove if the two ranges overlap.
cmoc.h · See it used in message.c

memichr

```
void *memichr(const void *s, int c, size_t n);
```

Search the first n bytes for c. The result points inside the original range, or is NULL. memichr folds letter case.
cmoc.h

memicmp

```
int memicmp(const void *s1, const void *s2, size_t n);
```

Compare exactly n bytes of two readable ranges, including zero bytes. Zero means a match. memicmp ignores ASCII case.
cmoc.h

memmove

```
void *memmove(void *dest, const void *src, size_t n);
```

Copy n bytes safely even if source and destination overlap, for example when shifting a row inside a manifest. Returns dest.

cmoc.h

memset

```
void *memset(void *s, int c, size_t n);
```

Fill n bytes with the low eight bits of c. Returns the start address.

cmoc.h · See it used in glyph.c

memset16

```
void *memset16(void *dest, unsigned short w, size_t num16BitWords);
```

Fill num16BitWords two-byte words. The destination needs twice that many bytes.

cmoc.h

mulwb

```
unsigned mulwb(unsigned char *hi, unsigned wordFactor, unsigned char byteFactor);
```

Multiply the two unsigned inputs. Use the returned low word and the high portion stored through hi to recover the full product.

cmoc.h

mulww

```
unsigned mulww(unsigned *hi, unsigned factor0, unsigned factor1);
```

Multiply the two unsigned inputs. Use the returned low word and the high portion stored through hi to recover the full product.

cmoc.h

NULL

```
NULL = ((void *) 0)
```

The null pointer value. Check it before using a lookup or allocation result as an address.

cmoc.h · See it used in tone.c

offsetof

```
offsetof(Type,member)
```

Retrieve the offset in bytes of a member within a structure type. Useful when comparing C data with the debugger's memory view.

cmoc.h

printf

```
int printf(const char *format, ...);
```

Send formatted characters to the current console. For the cargo total use %u with an unsigned word. Other useful conversions are %d, %x, %X, %p, %s, %c and %%. A width can pad a field; left justification applies to strings. CMOC also handles long integers with a length modifier.

cmoc.h · See it used in planet.c

putchar

```
void putchar(int c);
```

Write one character through the current console output hook.

cmoc.h

putstr

```
void putstr(const char *s, size_t n);
```

Send exactly n bytes. A zero byte inside the range does not shorten it. Keep all n bytes accessible.

cmoc.h

qsort

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

Supply the size of one element in bytes and a consistent three-way comparator. qsort reorders the array in place. bsearch expects that ordering and returns a pointer into the array or NULL.

cmoc.h

rand

```
int rand(void);
```

srand selects the starting seed for rand. Reusing a seed gives a repeatable sequence for a game or test. rand returns values from zero to RAND_MAX.

cmoc.h

RAND_MAX

```
RAND_MAX = 0x7FFF
```

srand selects the starting seed for rand. Reusing a seed gives a repeatable sequence for a game or test. rand returns values from zero to RAND_MAX.

cmoc.h

readline

```
char *readline(void);
```

Wait for a console line. readword converts a decimal word; readline returns a shared string or NULL on failure. Copy that string before a later read.

cmoc.h

readword

```
unsigned readword(void);
```

Wait for a console line. readword converts a decimal word; readline returns a shared string or NULL on failure. Copy that string before a later read.

cmoc.h

sbrk

```
void *sbrk(size_t increment);
```

Reserve bytes from the remaining target heap with sbrk. Failure is (void *)−1. A zero request queries the break; sbrkmax reports how many bytes are still available.

cmoc.h

sbrkmax

```
size_t sbrkmax(void);
```

Reserve bytes from the remaining target heap with sbrk. Failure is (void *)−1. A zero request queries the break; sbrkmax reports how many bytes are still available.

cmoc.h

set_null_ptr_handler

```
void set_null_ptr_handler(void (*newHandler)(void *addressPastFailedCheck));
```

With `--check-null`, install this callback before a checked pointer access can occur. In CMOC 0.1.101, `addressPastFailedCheck` is the address immediately after the call to the check routine, not the address of that

call instruction. Use the matching listing and map to locate the user function. The handler may return, but doing so can allow a dangerous null access to continue; a failure handler normally reports the problem and exits. This check is opt-in and adds code and execution cost.

cmoc.h

set_stack_overflow_handler

```
void set_stack_overflow_handler(void (*newHandler)(void *addressPastFailedCheck, void *stackRegister));
```

With `--check-stack`, install this callback before a checked function can overflow the stack. In CMOC 0.1.101, `addressPastFailedCheck` points immediately after the call to the check routine; `stackRegister` is the out-of-range stack pointer. The callback must not return. Use the matching listing to interpret the first address. This check is opt-in and adds execution cost.

cmoc.h

setConsoleOutHook

```
ConsoleOutHook setConsoleOutHook(ConsoleOutHook routine);
```

Install a character-output routine and keep the previous hook to restore later. The character arrives in register A. Preserve B, X, Y and U, including when the program calls printf.

cmoc.h

SIZE_MAX

```
SIZE_MAX = 0xFFFFu
```

A size in this target ABI occupies 16 bits. Unsigned size_t ranges from 0 to 65,535. Signed ssize_t tops out at 32,767.

cmoc.h

size_t

```
typedef unsigned size_t;
```

A size in this target ABI occupies 16 bits. Unsigned size_t ranges from 0 to 65,535. Signed ssize_t tops out at 32,767.

cmoc.h · See it used in message.c

sprintf

```
int sprintf(char *dest, const char *format, ...);
```

Write formatted characters and a final zero into the caller's array. There is no capacity argument. Allow space for the longest possible expansion and its terminator. Both routines share formatting state; avoid re-entering from an interrupt.

cmoc.h

sqrt16

```
unsigned char sqrt16(unsigned short n);
```

Return the integer part of the square root of an unsigned value. Use sqrt32 for a 32-bit input.

cmoc.h

sqrt32

```
unsigned short sqrt32(unsigned long n);
```

Return the integer part of the square root of an unsigned value. Use sqrt32 for a 32-bit input.

cmoc.h

srand

```
void srand(unsigned seed);
```

srand selects the starting seed for rand. Reusing a seed gives a repeatable sequence for a game or test. rand returns values from zero to RAND_MAX.

cmoc.h

SSIZE_MAX

```
SSIZE_MAX = 0x7FFF
```

A size in this target ABI occupies 16 bits. Unsigned size_t ranges from 0 to 65,535. Signed ssize_t tops out at 32,767.

cmoc.h

ssize_t

```
typedef signed ssize_t;
```

A size in this target ABI occupies 16 bits. Unsigned size_t ranges from 0 to 65,535. Signed ssize_t tops out at 32,767.

cmoc.h

strcat

```
char *strcat(char *dest, const char *src);
```

The destination needs enough room for the finished string and its zero byte. strcpy replaces its contents; strcat starts at the existing terminator. Both return dest.

cmoc.h

strchr

```
char *strchr(const char *s, int c);
```

Return a pointer into the original string, or NULL if there is no match. strchr finds the first character, strrchr the last, strstr a substring and strpbrk a character from the supplied set.

cmoc.h

strcmp

```
int strcmp(const char *s1, const char *s2);
```

Compare two zero-terminated strings. Zero means equal. A negative or positive result orders the strings. stricmp folds letter case.

cmoc.h · See it used in message.c

strcpy

```
char *strcpy(char *dest, const char *src);
```

The destination needs enough room for the finished string and its zero byte. strcpy replaces its contents; strcat starts at the existing terminator. Both return dest.

cmoc.h

strcspn

```
size_t strcspn(const char *s, const char *reject);
```

Count the initial run of characters drawn from accept (strspn), or outside reject (strcspn). The string is left intact.

cmoc.h

stricmp

```
int stricmp(const char *s1, const char *s2);
```

Compare two zero-terminated strings. Zero means equal. A negative or positive result orders the strings. stricmp folds letter case.

cmoc.h

strlen

```
size_t strlen(const char *s);
```

Count characters before the first zero. A 256-byte disk sector is only a C string once you establish a terminator within its buffer.

cmoc.h · See it used in message.c

strlwr

```
char *strlwr(char *s);
```

Change the case of ASCII letters in the supplied writable string. The result is the same storage.

cmoc.h

strncat

```
char *strncat(char *dest, const char *src, size_t n);
```

Append up to n source characters and terminate the result. n limits the source contribution; it does not specify the destination capacity.

cmoc.h

strncmp

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Compare up to n characters, stopping earlier at a string end. Zero means the compared prefixes match.

cmoc.h

strncpy

```
char *strncpy(char *dest, const char *src, size_t n);
```

Write n destination bytes, padding a short source with zeros. A source of length n or greater leaves the caller responsible for termination.

cmoc.h

strpbrk

```
char *strpbrk(const char *s, const char *accept);
```

Return a pointer into the original string, or NULL if there is no match. strchr finds the first character, strrchr the last, strstr a substring and strpbrk a character from the supplied set.

cmoc.h

strrchr

```
char *strrchr(const char *s, int c);
```

Return a pointer into the original string, or NULL if there is no match. strchr finds the first character, strrchr the last, strstr a substring and strpbrk a character from the supplied set.

cmoc.h

strspn

```
size_t strspn(const char *s, const char *accept);
```

Count the initial run of characters drawn from accept (strspn), or outside reject (strcspn). The string is left intact.

cmoc.h

strstr

```
char *strstr(const char *haystack, const char *needle);
```

Return a pointer into the original string, or NULL if there is no match. strchr finds the first character, strrchr the last, strstr a substring and strpbrk a character from the supplied set.

cmoc.h

strtof

```
float strtof(const char *nptr, char **endptr);
```

Parse a decimal floating-point number. In the BASIC backend use uppercase E for an exponent; spaces can occur within a number. strtof can report the first unconsumed character through endptr.

cmoc.h

strtok

```
char *strtok(char *str, const char *delim);
```

Split a writable string at any delimiter, replacing separators with zeros. Start with the string pointer, pass NULL to continue and finish when the returned pointer is NULL. Successive calls share a position. To work on a second string partway through the first, give each string its own position with strtok_r.

cmoc.h

strtol

```
strtol(nptr,endptr,base)
```

With the default CMOC headers, this macro calls **strtol10**. Its third argument is ignored. To select the base or check the 32-bit range, use the vcccompat function from **stdlib.h**.

cmoc.h

strtol10

```
strtol10(nptr,endptr)
```

Read decimal digits with the CMOC routine and interpret its result as a signed long. Supply **endptr** to find the first unconverted character. The vcccompat strtol function also checks the signed range.

cmoc.h

strtol16

```
strtol16(nptr,endptr)
```

Use the CMOC hexadecimal routine and interpret its result as a signed long. Supply **endptr** to find the first unconverted character. The vcccompat strtol function also checks the signed range.

cmoc.h

strtoul

```
strtoul(nptr,endptr,base)
```

With the default CMOC headers, this macro calls **strtoul10**. Its third argument is ignored. To select the base or check the 32-bit range, use the vcccompat function from **stdlib.h**.

cmoc.h

strtoul10

```
unsigned long strtoul10(const char *nptr, char **endptr);
```

Read an unsigned long in decimal using the CMOC runtime. Supply **endptr** to find the first unconverted character. Use the vcccompat strtoul function for base selection and range checking.

cmoc.h

strtoul16

```
unsigned long strtoul16(const char *nptr, char **endptr);
```

Read an unsigned long in hexadecimal using the CMOC runtime. Supply **endptr** to find the first unconverted character. Use the vcccompat strtoul function for base selection and range checking.

cmoc.h

strupr

```
char *strupr(char *s);
```

Change the case of ASCII letters in the supplied writable string. The result is the same storage.
cmoc.h

subdww

```
void subdww(unsigned *twoWords, unsigned term);
```

Treat two words, high word first, as one unsigned 32-bit quantity. The first three calls update it in place. Comparison returns −1, 0 or +1.
cmoc.h

tolower

```
int tolower(int c);
```

Return the converted character value; assign it back if you intend to change the stored text.
cmoc.h

toupper

```
int toupper(int c);
```

Return the converted character value; assign it back if you intend to change the stored text.
cmoc.h

ultoa

```
ultoa(value,str,base)
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

ultoa10

```
char *ultoa10(unsigned long value, char *str);
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

ultoa16

```
char *ultoa16(unsigned long value, char *str);
```

Convert a 32-bit unsigned value to hexadecimal and terminate the string. Reserve room for eight digits and the final zero.
cmoc.h

utoa

```
utoa(value,str,base)
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.
cmoc.h

utoa10

```
char *utoa10(unsigned value, char *str);
```

Write decimal digits and a terminating zero into str. Signed output may need a minus sign. Allow 7 bytes for a signed 16-bit value and 12 for a signed 32-bit value. The three-argument compatibility forms always use decimal.

cmoc.h

vprintf

```
int vprintf(const char *format, va_list ap);
```

Print using a va_list already positioned at the first variable argument. Use this in a wrapper that receives its own format and arguments.

cmoc.h

vsprintf

```
int vsprintf(char *dest, const char *format, va_list ap);
```

Write formatted characters and a final zero into the caller's array. There is no capacity argument. Allow space for the longest possible expansion and its terminator. Both routines share formatting state; avoid re-entering from an interrupt.

cmoc.h

zerodw

```
void zerodw(unsigned *twoWords);
```

Treat two words, high word first, as one unsigned 32-bit quantity. The first three calls update it in place. Comparison returns −1, 0 or +1.

cmoc.h

coco.h

BASIC display modes, keys, joysticks, sound and timing. The CoCo target supplies this header and runtime automatically.

One exception is readDECBFile, which requires the optional standalone controller driver. The WithDECB version uses Disk BASIC.

atanf · attr · BOOL · byte · cls · cmp · coldStart · cosf · DEV_CASSETTE · DEV_PRINTER · DEV_SCREEN · DEV_SERIAL · disableInterrupts · dword · enable6847T1TrueLowerCase · enableInterrupts · expf · fabsf · FALSE · floorf · getTextMode · getTimer · hscreen · hset · initCoCoSupport · inkey · int16_t · int32_t · int8_t · isCoCo3 · isKeyPressed · JOYSTK_BUTTON_1_LEFT · JOYSTK_BUTTON_1_RIGHT · JOYSTK_BUTTON_2_LEFT · JOYSTK_BUTTON_2_RIGHT · JOYSTK_LEFT_HORIZ · JOYSTK_LEFT_VERT · JOYSTK_MAX · JOYSTK_RIGHT_HORIZ · JOYSTK_RIGHT_VERT · KEY_BIT_0 · KEY_BIT_1 · KEY_BIT_2 · KEY_BIT_3 · KEY_BIT_4 · KEY_BIT_5 · KEY_BIT_6 · KEY_BIT_7 · KEY_BIT_8 · KEY_BIT_9 · KEY_BIT_A · KEY_BIT_ALT · KEY_BIT_AT · KEY_BIT_B · KEY_BIT_BREAK · KEY_BIT_C · KEY_BIT_CLEAR · KEY_BIT_COLON · KEY_BIT_COMMA · KEY_BIT_CTRL · KEY_BIT_D · KEY_BIT_DOWN · KEY_BIT_E · KEY_BIT_ENTER · KEY_BIT_F · KEY_BIT_F1 · KEY_BIT_F2 · KEY_BIT_G · KEY_BIT_H · KEY_BIT_HYPHEN · KEY_BIT_I · KEY_BIT_J · KEY_BIT_K · KEY_BIT_L · KEY_BIT_LEFT · KEY_BIT_M · KEY_BIT_N · KEY_BIT_O · KEY_BIT_P · KEY_BIT_PERIOD · KEY_BIT_Q · KEY_BIT_R · KEY_BIT_RIGHT · KEY_BIT_S · KEY_BIT_SEMICOLON · KEY_BIT_SHIFT · KEY_BIT_SLASH · KEY_BIT_SPACE · KEY_BIT_T · KEY_BIT_U · KEY_BIT_UP · KEY_BIT_V · KEY_BIT_W · KEY_BIT_X · KEY_BIT_Y · KEY_BIT_Z · KEY_PROBE_0 · KEY_PROBE_1 · KEY_PROBE_2 · KEY_PROBE_3 · KEY_PROBE_4 · KEY_PROBE_5 · KEY_PROBE_6 · KEY_PROBE_7 · KEY_PROBE_8 · KEY_PROBE_9 · KEY_PROBE_A · KEY_PROBE_ALT · KEY_PROBE_AT · KEY_PROBE_B · KEY_PROBE_BREAK · KEY_PROBE_C · KEY_PROBE_CLEAR · KEY_PROBE_COLON · KEY_PROBE_COMMA · KEY_PROBE_CTRL · KEY_PROBE_D · KEY_PROBE_DOWN · KEY_PROBE_E · KEY_PROBE_ENTER · KEY_PROBE_F · KEY_PROBE_F1 · KEY_PROBE_F2 · KEY_PROBE_G · KEY_PROBE_H · KEY_PROBE_HYPHEN · KEY_PROBE_I · KEY_PROBE_J · KEY_PROBE_K · KEY_PROBE_L · KEY_PROBE_LEFT · KEY_PROBE_M · KEY_PROBE_N · KEY_PROBE_O · KEY_PROBE_P · KEY_PROBE_PERIOD · KEY_PROBE_Q · KEY_PROBE_R · KEY_PROBE_RIGHT · KEY_PROBE_S · KEY_PROBE_SEMICOLON · KEY_PROBE_SHIFT · KEY_PROBE_SLASH · KEY_PROBE_SPACE · KEY_PROBE_T · KEY_PROBE_U · KEY_PROBE_UP · KEY_PROBE_V · KEY_PROBE_W · KEY_PROBE_X · KEY_PROBE_Y · KEY_PROBE_Z · KeyboardBits · locate · palette · paletteRGB · pcls · pmode · readDECBFile · readDECBFileWithDECB ·

readJoystickButtons · readJoystickPositions · resetPalette · rgb · roundf · sbyte · screen · sdword · setBorderColor · setCaseFlag · setHighSpeed · setOutputDevice · setPmodeGraphicsAddress · setPmodeGraphicsAddressEx · setSAMRegisters · setTimer · showGraphicsAddress · showLowResTextAddress · showPmode4 · sinf · sleep · sound · sqrtf · sword · tanf · textScreenHeight · textScreenWidth · TRUE · truncf · uint16_t · uint32_t · uint8_t · waitkey · width · word

atanf

```
float atanf(float radians);
```

The trigonometric functions use radians. Tangent is undefined at odd multiples of π/2. The BASIC ROM computes these results in the standard CoCo configuration.

coco.h

attr

```
byte attr(byte foreColor, byte backColor, byte blink, byte underline);
```

Set the color indices for subsequent CoCo 3 text, plus blink and underline flags. Foreground and background each take 0–7. The hardware uses foreground palette slots 8–15 and background slots 0–7.

coco.h · See it used in planet.c

BOOL

```
typedef unsigned char BOOL;
```

A BOOL is one byte. A zero value tests false and any nonzero value tests true. Use named constants when passing option flags.

coco.h

byte

```
typedef unsigned char byte;
```

Eight-bit storage. Unsigned values occupy 0–255 and signed values −128–127. Cargo Count uses byte counts for crates and meals per crate.

coco.h · See it used in planet.c

cls

```
void cls(byte color);
```

Clear the BASIC text screen. A value of 255 requests the no-argument BASIC CLS behavior.

coco.h · See it used in editlog.c

cmp

```
void cmp(void);
```

Load the standard RGB or composite color assignments for the CoCo 3 palette. Select a matching signal in the emulator if comparing the two color systems.

coco.h

coldStart

```
void coldStart();
```

Restart BASIC through its cold-start entry. This abandons the running program.

coco.h

cosf

```
float cosf(float radians);
```

The trigonometric functions use radians. Tangent is undefined at odd multiples of π/2. The BASIC ROM computes these results in the standard CoCo configuration.

coco.h

DEV_CASSETTE

```
DEV_CASSETTE = -1
```

A BASIC output destination for setOutputDevice.

coco.h

DEV_PRINTER

```
DEV_PRINTER = -2
```

A BASIC output destination for setOutputDevice.

coco.h

DEV_SCREEN

```
DEV_SCREEN = 0
```

A BASIC output destination for setOutputDevice.

coco.h

DEV_SERIAL

```
DEV_SERIAL = -2
```

A BASIC output destination for setOutputDevice.

coco.h

disableInterrupts

```
disableInterrupts()
```

Mask or enable both IRQ and FIRQ. Leave interrupt-dependent keyboard, disk and audio routines with the services they require.

coco.h

dword

```
typedef unsigned long dword;
```

Thirty-two-bit storage. Good for tracking absolute file lengths that exceed 65,535 bytes. The debugger shows all four bytes of the guest value.

coco.h

enable6847T1TrueLowerCase

```
enable6847T1TrueLowerCase()
```

Enable lowercase character shapes in the compatible 32-column text mode. BASIC character output can change this setting again.

coco.h

enableInterrupts

```
enableInterrupts()
```

Mask or enable both IRQ and FIRQ. Leave interrupt-dependent keyboard, disk and audio routines with the services they require.

coco.h

expf

```
float expf(float x);
```

These functions use the selected floating-point backend. For the BASIC backend keep its ROM mapped. logf requires a positive argument and sqrtf requires zero or greater; a BASIC math error can end the program.

coco.h

fabsf

```
float fabsf(float x);
```

These functions use the selected floating-point backend. For the BASIC backend keep its ROM mapped. logf requires a positive argument and sqrtf requires zero or greater; a BASIC math error can end the program.

coco.h

FALSE

```
FALSE = 0
```

A BOOL is one byte. A zero value tests false and any nonzero value tests true. Use named constants when passing option flags.

coco.h · See it used in planet.c

floorf

```
float floorf(float x);
```

Return a floating-point integral value: floor goes toward negative infinity, trunc toward zero, and round to nearest with exact halves away from zero.

coco.h

getTextMode

```
byte getTextMode();
```

Under BASIC on the CoCo 3 choose 32, 40 or 80 columns. width reports success or failure. getTextMode reports the actual width.

coco.h

getTimer

```
getTimer()
```

Access the BASIC 16-bit tick count. It wraps after 65,535; setting it changes the time seen by other guest code.

coco.h · See it used in stick.c

hscreen

```
byte hscreen(byte mode);
```

Select a CoCo 3 BASIC graphics mode before plotting. hset takes a pixel address and a color in the BASIC mode's allowed range. These ROM calls depend on BASIC's graphics workspace.

coco.h · See it used in beacon.c

hset

```
byte hset(word x, word y, byte color);
```

Select a CoCo 3 BASIC graphics mode before plotting. hset takes a pixel address and a color in the BASIC mode's allowed range. These ROM calls depend on BASIC's graphics workspace.

coco.h · See it used in beacon.c

initCoCoSupport

```
void initCoCoSupport(void);
```

Legacy program state populated by initCoCoSupport. Set the required mode explicitly with width and read the mode through getTextMode for new projects.

coco.h · See it used in main.c

inkey

```
byte inkey();
```

inkey returns zero while no character is ready. waitkey waits and accepts a cursor-blink flag. Both use the BASIC keyboard routine.

coco.h · See it used in planet.c

int16_t

```
typedef signed int int16_t;
```

Sixteen-bit storage. An unsigned word can hold 360 even though an eight-bit byte cannot. Signed values span −32,768 to 32,767.

coco.h

int32_t

```
typedef signed long int32_t;
```

Thirty-two-bit storage. Good for tracking absolute file lengths that exceed 65,535 bytes. The debugger shows all four bytes of the guest value.

coco.h

int8_t

```
typedef signed char int8_t;
```

Eight-bit storage. Unsigned values occupy 0–255 and signed values −128–127. Cargo Count uses byte counts for crates and meals per crate.

coco.h

isCoCo3

```
extern byte isCoCo3;
```

Legacy program state populated by initCoCoSupport. Set the required mode explicitly with width and read the mode through getTextMode for new projects.

coco.h

isKeyPressed

```
byte isKeyPressed(byte probe, byte testBit);
```

Supply the matching KEY_PROBE and KEY_BIT values for one key. This polls the CoCo keyboard matrix without consuming a console character.

coco.h

JOYSTK_BUTTON_1_LEFT

```
JOYSTK_BUTTON_1_LEFT = 2
```

A mask for one active-low button in the value from readJoystickButtons.

coco.h

JOYSTK_BUTTON_1_RIGHT

```
JOYSTK_BUTTON_1_RIGHT = 1
```

A mask for one active-low button in the value from readJoystickButtons.

coco.h

JOYSTK_BUTTON_2_LEFT

```
JOYSTK_BUTTON_2_LEFT = 8
```

A mask for one active-low button in the value from readJoystickButtons.

coco.h

JOYSTK_BUTTON_2_RIGHT

```
JOYSTK_BUTTON_2_RIGHT = 4
```

A mask for one active-low button in the value from readJoystickButtons.

coco.h

JOYSTK_LEFT_HORIZ

```
JOYSTK_LEFT_HORIZ = 2
```

An axis index or range limit for the array returned by readJoystickPositions.

coco.h

JOYSTK_LEFT_VERT

```
JOYSTK_LEFT_VERT = 3
```

An axis index or range limit for the array returned by readJoystickPositions.

coco.h

JOYSTK_MAX

```
JOYSTK_MAX = 63
```

An axis index or range limit for the array returned by readJoystickPositions.

coco.h

JOYSTK_RIGHT_HORIZ

```
JOYSTK_RIGHT_HORIZ = 0
```

An axis index or range limit for the array returned by readJoystickPositions.

coco.h

JOYSTK_RIGHT_VERT

```
JOYSTK_RIGHT_VERT = 1
```

An axis index or range limit for the array returned by readJoystickPositions.

coco.h

KEY_BIT_0

```
KEY_BIT_0 = 16
```

Keyboard matrix value for 0. Pass KEY_PROBE_0 and KEY_BIT_0 together to isKeyPressed.

coco.h

KEY_BIT_1

```
KEY_BIT_1 = 16
```

Keyboard matrix value for 1. Pass KEY_PROBE_1 and KEY_BIT_1 together to isKeyPressed.

coco.h

KEY_BIT_2

```
KEY_BIT_2 = 16
```

Keyboard matrix value for 2. Pass KEY_PROBE_2 and KEY_BIT_2 together to isKeyPressed.

coco.h

KEY_BIT_3

```
KEY_BIT_3 = 16
```

Keyboard matrix value for 3. Pass KEY_PROBE_3 and KEY_BIT_3 together to isKeyPressed.

coco.h

KEY_BIT_4

```
KEY_BIT_4 = 16
```

Keyboard matrix value for 4. Pass KEY_PROBE_4 and KEY_BIT_4 together to isKeyPressed.

coco.h

KEY_BIT_5

```
KEY_BIT_5 = 16
```

Keyboard matrix value for 5. Pass KEY_PROBE_5 and KEY_BIT_5 together to isKeyPressed.

coco.h

KEY_BIT_6

```
KEY_BIT_6 = 16
```

Keyboard matrix value for 6. Pass KEY_PROBE_6 and KEY_BIT_6 together to isKeyPressed.

coco.h

KEY_BIT_7

```
KEY_BIT_7 = 16
```

Keyboard matrix value for 7. Pass KEY_PROBE_7 and KEY_BIT_7 together to isKeyPressed.

coco.h

KEY_BIT_8

```
KEY_BIT_8 = 32
```

Keyboard matrix value for 8. Pass KEY_PROBE_8 and KEY_BIT_8 together to isKeyPressed.

coco.h

KEY_BIT_9

```
KEY_BIT_9 = 32
```

Keyboard matrix value for 9. Pass KEY_PROBE_9 and KEY_BIT_9 together to isKeyPressed.

coco.h

KEY_BIT_A

```
KEY_BIT_A = 1
```

Keyboard matrix value for A. Pass KEY_PROBE_A and KEY_BIT_A together to isKeyPressed.

coco.h

KEY_BIT_ALT

```
KEY_BIT_ALT = 64
```

Keyboard matrix value for ALT. Pass KEY_PROBE_ALT and KEY_BIT_ALT together to isKeyPressed.

coco.h

KEY_BIT_AT

```
KEY_BIT_AT = 1
```

Keyboard matrix value for AT. Pass KEY_PROBE_AT and KEY_BIT_AT together to isKeyPressed.

coco.h

KEY_BIT_B

```
KEY_BIT_B = 1
```

Keyboard matrix value for B. Pass KEY_PROBE_B and KEY_BIT_B together to isKeyPressed.

coco.h

KEY_BIT_BREAK

```
KEY_BIT_BREAK = 64
```

Keyboard matrix value for BREAK. Pass KEY_PROBE_BREAK and KEY_BIT_BREAK together to isKeyPressed.

coco.h

KEY_BIT_C

```
KEY_BIT_C = 1
```

Keyboard matrix value for C. Pass KEY_PROBE_C and KEY_BIT_C together to isKeyPressed.

coco.h

KEY_BIT_CLEAR

```
KEY_BIT_CLEAR = 64
```

Keyboard matrix value for CLEAR. Pass KEY_PROBE_CLEAR and KEY_BIT_CLEAR together to isKeyPressed.

coco.h

KEY_BIT_COLON

```
KEY_BIT_COLON = 32
```

Keyboard matrix value for COLON. Pass KEY_PROBE_COLON and KEY_BIT_COLON together to isKeyPressed.

coco.h

KEY_BIT_COMMA

```
KEY_BIT_COMMA = 32
```

Keyboard matrix value for COMMA. Pass KEY_PROBE_COMMA and KEY_BIT_COMMA together to isKeyPressed.

coco.h

KEY_BIT_CTRL

```
KEY_BIT_CTRL = 64
```

Keyboard matrix value for CTRL. Pass KEY_PROBE_CTRL and KEY_BIT_CTRL together to isKeyPressed.

coco.h

KEY_BIT_D

```
KEY_BIT_D = 1
```

Keyboard matrix value for D. Pass KEY_PROBE_D and KEY_BIT_D together to isKeyPressed.

coco.h

KEY_BIT_DOWN

```
KEY_BIT_DOWN = 8
```

Keyboard matrix value for DOWN. Pass KEY_PROBE_DOWN and KEY_BIT_DOWN together to isKeyPressed.

coco.h

KEY_BIT_E

```
KEY_BIT_E = 1
```

Keyboard matrix value for E. Pass KEY_PROBE_E and KEY_BIT_E together to isKeyPressed.

coco.h

KEY_BIT_ENTER

```
KEY_BIT_ENTER = 64
```

Keyboard matrix value for ENTER. Pass KEY_PROBE_ENTER and KEY_BIT_ENTER together to isKeyPressed.

coco.h

KEY_BIT_F

```
KEY_BIT_F = 1
```

Keyboard matrix value for F. Pass KEY_PROBE_F and KEY_BIT_F together to isKeyPressed.

coco.h

KEY_BIT_F1

`KEY_BIT_F1 = 64`

Keyboard matrix value for F1. Pass KEY_PROBE_F1 and KEY_BIT_F1 together to isKeyPressed.

coco.h

KEY_BIT_F2

`KEY_BIT_F2 = 64`

Keyboard matrix value for F2. Pass KEY_PROBE_F2 and KEY_BIT_F2 together to isKeyPressed.

coco.h

KEY_BIT_G

`KEY_BIT_G = 1`

Keyboard matrix value for G. Pass KEY_PROBE_G and KEY_BIT_G together to isKeyPressed.

coco.h

KEY_BIT_H

`KEY_BIT_H = 2`

Keyboard matrix value for H. Pass KEY_PROBE_H and KEY_BIT_H together to isKeyPressed.

coco.h

KEY_BIT_HYPHEN

`KEY_BIT_HYPHEN = 32`

Keyboard matrix value for HYPHEN. Pass KEY_PROBE_HYPHEN and KEY_BIT_HYPHEN together to isKeyPressed.

coco.h

KEY_BIT_I

`KEY_BIT_I = 2`

Keyboard matrix value for I. Pass KEY_PROBE_I and KEY_BIT_I together to isKeyPressed.

coco.h

KEY_BIT_J

`KEY_BIT_J = 2`

Keyboard matrix value for J. Pass KEY_PROBE_J and KEY_BIT_J together to isKeyPressed.

coco.h

KEY_BIT_K

`KEY_BIT_K = 2`

Keyboard matrix value for K. Pass KEY_PROBE_K and KEY_BIT_K together to isKeyPressed.

coco.h

KEY_BIT_L

`KEY_BIT_L = 2`

Keyboard matrix value for L. Pass KEY_PROBE_L and KEY_BIT_L together to isKeyPressed.

coco.h

KEY_BIT_LEFT

`KEY_BIT_LEFT = 8`

Keyboard matrix value for LEFT. Pass KEY_PROBE_LEFT and KEY_BIT_LEFT together to isKeyPressed.

coco.h

KEY_BIT_M

`KEY_BIT_M = 2`

Keyboard matrix value for M. Pass KEY_PROBE_M and KEY_BIT_M together to isKeyPressed.

coco.h

KEY_BIT_N

`KEY_BIT_N = 2`

Keyboard matrix value for N. Pass KEY_PROBE_N and KEY_BIT_N together to isKeyPressed.

coco.h

KEY_BIT_O

`KEY_BIT_O = 2`

Keyboard matrix value for O. Pass KEY_PROBE_O and KEY_BIT_O together to isKeyPressed.

coco.h

KEY_BIT_P

`KEY_BIT_P = 4`

Keyboard matrix value for P. Pass KEY_PROBE_P and KEY_BIT_P together to isKeyPressed.

coco.h

KEY_BIT_PERIOD

`KEY_BIT_PERIOD = 32`

Keyboard matrix value for PERIOD. Pass KEY_PROBE_PERIOD and KEY_BIT_PERIOD together to isKeyPressed.

coco.h

KEY_BIT_Q

`KEY_BIT_Q = 4`

Keyboard matrix value for Q. Pass KEY_PROBE_Q and KEY_BIT_Q together to isKeyPressed.

coco.h

KEY_BIT_R

`KEY_BIT_R = 4`

Keyboard matrix value for R. Pass KEY_PROBE_R and KEY_BIT_R together to isKeyPressed.

coco.h

KEY_BIT_RIGHT

`KEY_BIT_RIGHT = 8`

Keyboard matrix value for RIGHT. Pass KEY_PROBE_RIGHT and KEY_BIT_RIGHT together to isKeyPressed.

coco.h

KEY_BIT_S

`KEY_BIT_S = 4`

Keyboard matrix value for S. Pass KEY_PROBE_S and KEY_BIT_S together to isKeyPressed.

coco.h

KEY_BIT_SEMICOLON

`KEY_BIT_SEMICOLON = 32`

Keyboard matrix value for SEMICOLON. Pass KEY_PROBE_SEMICOLON and KEY_BIT_SEMICOLON together to isKeyPressed.

coco.h

KEY_BIT_SHIFT

`KEY_BIT_SHIFT = 64`

Keyboard matrix value for SHIFT. Pass KEY_PROBE_SHIFT and KEY_BIT_SHIFT together to isKeyPressed.

coco.h

KEY_BIT_SLASH

`KEY_BIT_SLASH = 32`

Keyboard matrix value for SLASH. Pass KEY_PROBE_SLASH and KEY_BIT_SLASH together to isKeyPressed.

coco.h

KEY_BIT_SPACE

`KEY_BIT_SPACE = 8`

Keyboard matrix value for SPACE. Pass KEY_PROBE_SPACE and KEY_BIT_SPACE together to isKeyPressed.

coco.h

KEY_BIT_T

`KEY_BIT_T = 4`

Keyboard matrix value for T. Pass KEY_PROBE_T and KEY_BIT_T together to isKeyPressed.

coco.h

KEY_BIT_U

`KEY_BIT_U = 4`

Keyboard matrix value for U. Pass KEY_PROBE_U and KEY_BIT_U together to isKeyPressed.

coco.h

KEY_BIT_UP

`KEY_BIT_UP = 8`

Keyboard matrix value for UP. Pass KEY_PROBE_UP and KEY_BIT_UP together to isKeyPressed.

coco.h

KEY_BIT_V

`KEY_BIT_V = 4`

Keyboard matrix value for V. Pass KEY_PROBE_V and KEY_BIT_V together to isKeyPressed.

coco.h

KEY_BIT_W

`KEY_BIT_W = 4`

Keyboard matrix value for W. Pass KEY_PROBE_W and KEY_BIT_W together to isKeyPressed.

coco.h

KEY_BIT_X

`KEY_BIT_X = 8`

Keyboard matrix value for X. Pass KEY_PROBE_X and KEY_BIT_X together to isKeyPressed.

coco.h

KEY_BIT_Y

`KEY_BIT_Y = 8`

Keyboard matrix value for Y. Pass KEY_PROBE_Y and KEY_BIT_Y together to isKeyPressed.

coco.h

KEY_BIT_Z

```
KEY_BIT_Z = 8
```

Keyboard matrix value for Z. Pass KEY_PROBE_Z and KEY_BIT_Z together to isKeyPressed.

coco.h

KEY_PROBE_0

```
KEY_PROBE_0 = 254
```

Keyboard matrix value for 0. Pass KEY_PROBE_0 and KEY_BIT_0 together to isKeyPressed.

coco.h

KEY_PROBE_1

```
KEY_PROBE_1 = 253
```

Keyboard matrix value for 1. Pass KEY_PROBE_1 and KEY_BIT_1 together to isKeyPressed.

coco.h

KEY_PROBE_2

```
KEY_PROBE_2 = 251
```

Keyboard matrix value for 2. Pass KEY_PROBE_2 and KEY_BIT_2 together to isKeyPressed.

coco.h

KEY_PROBE_3

```
KEY_PROBE_3 = 247
```

Keyboard matrix value for 3. Pass KEY_PROBE_3 and KEY_BIT_3 together to isKeyPressed.

coco.h

KEY_PROBE_4

```
KEY_PROBE_4 = 239
```

Keyboard matrix value for 4. Pass KEY_PROBE_4 and KEY_BIT_4 together to isKeyPressed.

coco.h

KEY_PROBE_5

```
KEY_PROBE_5 = 223
```

Keyboard matrix value for 5. Pass KEY_PROBE_5 and KEY_BIT_5 together to isKeyPressed.

coco.h

KEY_PROBE_6

```
KEY_PROBE_6 = 191
```

Keyboard matrix value for 6. Pass KEY_PROBE_6 and KEY_BIT_6 together to isKeyPressed.

coco.h

KEY_PROBE_7

```
KEY_PROBE_7 = 127
```

Keyboard matrix value for 7. Pass KEY_PROBE_7 and KEY_BIT_7 together to isKeyPressed.

coco.h

KEY_PROBE_8

```
KEY_PROBE_8 = 254
```

Keyboard matrix value for 8. Pass KEY_PROBE_8 and KEY_BIT_8 together to isKeyPressed.

coco.h

KEY_PROBE_9

`KEY_PROBE_9 = 253`

Keyboard matrix value for 9. Pass KEY_PROBE_9 and KEY_BIT_9 together to isKeyPressed.

coco.h

KEY_PROBE_A

`KEY_PROBE_A = 253`

Keyboard matrix value for A. Pass KEY_PROBE_A and KEY_BIT_A together to isKeyPressed.

coco.h

KEY_PROBE_ALT

`KEY_PROBE_ALT = 247`

Keyboard matrix value for ALT. Pass KEY_PROBE_ALT and KEY_BIT_ALT together to isKeyPressed.

coco.h

KEY_PROBE_AT

`KEY_PROBE_AT = 254`

Keyboard matrix value for AT. Pass KEY_PROBE_AT and KEY_BIT_AT together to isKeyPressed.

coco.h

KEY_PROBE_B

`KEY_PROBE_B = 251`

Keyboard matrix value for B. Pass KEY_PROBE_B and KEY_BIT_B together to isKeyPressed.

coco.h

KEY_PROBE_BREAK

`KEY_PROBE_BREAK = 251`

Keyboard matrix value for BREAK. Pass KEY_PROBE_BREAK and KEY_BIT_BREAK together to isKeyPressed.

coco.h

KEY_PROBE_C

`KEY_PROBE_C = 247`

Keyboard matrix value for C. Pass KEY_PROBE_C and KEY_BIT_C together to isKeyPressed.

coco.h

KEY_PROBE_CLEAR

`KEY_PROBE_CLEAR = 253`

Keyboard matrix value for CLEAR. Pass KEY_PROBE_CLEAR and KEY_BIT_CLEAR together to isKeyPressed.

coco.h

KEY_PROBE_COLON

`KEY_PROBE_COLON = 251`

Keyboard matrix value for COLON. Pass KEY_PROBE_COLON and KEY_BIT_COLON together to isKeyPressed.

coco.h

KEY_PROBE_COMMA

`KEY_PROBE_COMMA = 239`

Keyboard matrix value for COMMA. Pass KEY_PROBE_COMMA and KEY_BIT_COMMA together to isKeyPressed.

coco.h

KEY_PROBE_CTRL

`KEY_PROBE_CTRL = 239`

Keyboard matrix value for CTRL. Pass KEY_PROBE_CTRL and KEY_BIT_CTRL together to isKeyPressed.

coco.h

KEY_PROBE_D

`KEY_PROBE_D = 239`

Keyboard matrix value for D. Pass KEY_PROBE_D and KEY_BIT_D together to isKeyPressed.

coco.h

KEY_PROBE_DOWN

`KEY_PROBE_DOWN = 239`

Keyboard matrix value for DOWN. Pass KEY_PROBE_DOWN and KEY_BIT_DOWN together to isKeyPressed.

coco.h

KEY_PROBE_E

`KEY_PROBE_E = 223`

Keyboard matrix value for E. Pass KEY_PROBE_E and KEY_BIT_E together to isKeyPressed.

coco.h

KEY_PROBE_ENTER

`KEY_PROBE_ENTER = 254`

Keyboard matrix value for ENTER. Pass KEY_PROBE_ENTER and KEY_BIT_ENTER together to isKeyPressed.

coco.h

KEY_PROBE_F

`KEY_PROBE_F = 191`

Keyboard matrix value for F. Pass KEY_PROBE_F and KEY_BIT_F together to isKeyPressed.

coco.h

KEY_PROBE_F1

`KEY_PROBE_F1 = 223`

Keyboard matrix value for F1. Pass KEY_PROBE_F1 and KEY_BIT_F1 together to isKeyPressed.

coco.h

KEY_PROBE_F2

`KEY_PROBE_F2 = 191`

Keyboard matrix value for F2. Pass KEY_PROBE_F2 and KEY_BIT_F2 together to isKeyPressed.

coco.h

KEY_PROBE_G

`KEY_PROBE_G = 127`

Keyboard matrix value for G. Pass KEY_PROBE_G and KEY_BIT_G together to isKeyPressed.

coco.h

KEY_PROBE_H

`KEY_PROBE_H = 254`

Keyboard matrix value for H. Pass KEY_PROBE_H and KEY_BIT_H together to isKeyPressed.

coco.h

KEY_PROBE_HYPHEN

```
KEY_PROBE_HYPHEN = 223
```

Keyboard matrix value for HYPHEN. Pass KEY_PROBE_HYPHEN and KEY_BIT_HYPHEN together to isKeyPressed.

coco.h

KEY_PROBE_I

```
KEY_PROBE_I = 253
```

Keyboard matrix value for I. Pass KEY_PROBE_I and KEY_BIT_I together to isKeyPressed.

coco.h

KEY_PROBE_J

```
KEY_PROBE_J = 251
```

Keyboard matrix value for J. Pass KEY_PROBE_J and KEY_BIT_J together to isKeyPressed.

coco.h

KEY_PROBE_K

```
KEY_PROBE_K = 247
```

Keyboard matrix value for K. Pass KEY_PROBE_K and KEY_BIT_K together to isKeyPressed.

coco.h

KEY_PROBE_L

```
KEY_PROBE_L = 239
```

Keyboard matrix value for L. Pass KEY_PROBE_L and KEY_BIT_L together to isKeyPressed.

coco.h

KEY_PROBE_LEFT

```
KEY_PROBE_LEFT = 223
```

Keyboard matrix value for LEFT. Pass KEY_PROBE_LEFT and KEY_BIT_LEFT together to isKeyPressed.

coco.h

KEY_PROBE_M

```
KEY_PROBE_M = 223
```

Keyboard matrix value for M. Pass KEY_PROBE_M and KEY_BIT_M together to isKeyPressed.

coco.h

KEY_PROBE_N

```
KEY_PROBE_N = 191
```

Keyboard matrix value for N. Pass KEY_PROBE_N and KEY_BIT_N together to isKeyPressed.

coco.h

KEY_PROBE_O

```
KEY_PROBE_0 = 127
```

Keyboard matrix value for O. Pass KEY_PROBE_O and KEY_BIT_O together to isKeyPressed.

coco.h

KEY_PROBE_P

```
KEY_PROBE_P = 254
```

Keyboard matrix value for P. Pass KEY_PROBE_P and KEY_BIT_P together to isKeyPressed.

coco.h

KEY_PROBE_PERIOD

`KEY_PROBE_PERIOD = 191`

Keyboard matrix value for PERIOD. Pass KEY_PROBE_PERIOD and KEY_BIT_PERIOD together to isKeyPressed.

coco.h

KEY_PROBE_Q

`KEY_PROBE_Q = 253`

Keyboard matrix value for Q. Pass KEY_PROBE_Q and KEY_BIT_Q together to isKeyPressed.

coco.h

KEY_PROBE_R

`KEY_PROBE_R = 251`

Keyboard matrix value for R. Pass KEY_PROBE_R and KEY_BIT_R together to isKeyPressed.

coco.h

KEY_PROBE_RIGHT

`KEY_PROBE_RIGHT = 191`

Keyboard matrix value for RIGHT. Pass KEY_PROBE_RIGHT and KEY_BIT_RIGHT together to isKeyPressed.

coco.h

KEY_PROBE_S

`KEY_PROBE_S = 247`

Keyboard matrix value for S. Pass KEY_PROBE_S and KEY_BIT_S together to isKeyPressed.

coco.h

KEY_PROBE_SEMICOLON

`KEY_PROBE_SEMICOLON = 247`

Keyboard matrix value for SEMICOLON. Pass KEY_PROBE_SEMICOLON and KEY_BIT_SEMICOLON together to isKeyPressed.

coco.h

KEY_PROBE_SHIFT

`KEY_PROBE_SHIFT = 127`

Keyboard matrix value for SHIFT. Pass KEY_PROBE_SHIFT and KEY_BIT_SHIFT together to isKeyPressed.

coco.h

KEY_PROBE_SLASH

`KEY_PROBE_SLASH = 127`

Keyboard matrix value for SLASH. Pass KEY_PROBE_SLASH and KEY_BIT_SLASH together to isKeyPressed.

coco.h

KEY_PROBE_SPACE

`KEY_PROBE_SPACE = 127`

Keyboard matrix value for SPACE. Pass KEY_PROBE_SPACE and KEY_BIT_SPACE together to isKeyPressed.

coco.h

KEY_PROBE_T

`KEY_PROBE_T = 239`

Keyboard matrix value for T. Pass KEY_PROBE_T and KEY_BIT_T together to isKeyPressed.

coco.h

KEY_PROBE_U

```
KEY_PROBE_U = 223
```

Keyboard matrix value for U. Pass KEY_PROBE_U and KEY_BIT_U together to isKeyPressed.

coco.h

KEY_PROBE_UP

```
KEY_PROBE_UP = 247
```

Keyboard matrix value for UP. Pass KEY_PROBE_UP and KEY_BIT_UP together to isKeyPressed.

coco.h

KEY_PROBE_V

```
KEY_PROBE_V = 191
```

Keyboard matrix value for V. Pass KEY_PROBE_V and KEY_BIT_V together to isKeyPressed.

coco.h

KEY_PROBE_W

```
KEY_PROBE_W = 127
```

Keyboard matrix value for W. Pass KEY_PROBE_W and KEY_BIT_W together to isKeyPressed.

coco.h

KEY_PROBE_X

```
KEY_PROBE_X = 254
```

Keyboard matrix value for X. Pass KEY_PROBE_X and KEY_BIT_X together to isKeyPressed.

coco.h

KEY_PROBE_Y

```
KEY_PROBE_Y = 253
```

Keyboard matrix value for Y. Pass KEY_PROBE_Y and KEY_BIT_Y together to isKeyPressed.

coco.h

KEY_PROBE_Z

```
KEY_PROBE_Z = 251
```

Keyboard matrix value for Z. Pass KEY_PROBE_Z and KEY_BIT_Z together to isKeyPressed.

coco.h

KeyboardBits

```
enum KeyboardBits
```

Supply the matching KEY_PROBE and KEY_BIT values for one key. This polls the CoCo keyboard matrix without consuming a console character.

coco.h

locate

```
byte locate(byte column, byte row);
```

Set the printing position in BASIC text coordinates, starting at column 0 and row 0. Stay inside the current text dimensions.

coco.h

palette

```
byte palette(byte slot, byte color);
```

Set one of 16 CoCo 3 palette slots. palette accepts a hardware color 0–63 and reports whether its arguments were valid. paletteRGB uses separate red, green and blue components of 0–3 each.

coco.h · See it used in planet.c

paletteRGB

```
void paletteRGB(byte slot, byte red, byte green, byte blue);
```

Set one of 16 CoCo 3 palette slots. palette accepts a hardware color 0–63 and reports whether its arguments were valid. paletteRGB uses separate red, green and blue components of 0–3 each.

coco.h · See it used in beacon.c

pcls

```
void pcls(byte byteToClearWith);
```

pmode records a graphics mode from 0–4 and a 512-byte-aligned buffer; it reports invalid inputs. pcls fills that buffer with a **byte pattern**. screen selects text or the prepared graphics display and color set.

coco.h · See it used in glyph.c

pmode

```
BOOL pmode(byte mode, byte *screenBuffer);
```

pmode records a graphics mode from 0–4 and a 512-byte-aligned buffer; it reports invalid inputs. pcls fills that buffer with a **byte pattern**. screen selects text or the prepared graphics display and color set.

coco.h · See it used in glyph.c

readDECBFile

```
byte readDECBFile(void *dest, byte driveNo, const char filename[11], byte workBuffer[256], size_t *sizePtr);
```

Supply an eleven-character, space-padded 8+3 name, a separate 256-byte work buffer and a destination large enough for whole sectors. Return 0 signals success; 2 means missing file. readDECBFile takes over the interrupt vectors temporarily and returns with interrupts masked. The WithDECB variant needs the ROM and ordinary IRQ/NMI services. This entry also needs the optional dskcon-standalone archive at link time.

coco.h · `-ldskcon-standalone`

readDECBFileWithDECB

```
byte readDECBFileWithDECB(void *dest, byte driveNo, const char filename[11], byte workBuffer[256], size_t *sizePtr);
```

Supply an eleven-character, space-padded 8+3 name, a separate 256-byte work buffer and a destination large enough for whole sectors. Return 0 signals success; 2 means missing file. readDECBFile takes over the interrupt vectors temporarily and returns with interrupts masked. The WithDECB variant needs the ROM and ordinary IRQ/NMI services.

coco.h

readJoystickButtons

```
byte readJoystickButtons();
```

Read the raw low nibble of the CoCo PIA. Each cleared bit indicates a pressed button. The JOYSTK_BUTTON_* constants in coco.h give masks for the guest ports.

coco.h

readJoystickPositions

```
const byte *readJoystickPositions();
```

Poll the four axes; the returned pointer refers to four shared byte values from 0 to 63. Right horizontal/vertical are entries 0/1, left horizontal/vertical 2/3.

coco.h

resetPalette

```
byte resetPalette(byte isRGB);
```

Load the standard RGB or composite color assignments for the CoCo 3 palette. Select a matching signal in the emulator if comparing the two color systems.
coco.h

rgb

```
void rgb(void);
```

Load the standard RGB or composite color assignments for the CoCo 3 palette. Select a matching signal in the emulator if comparing the two color systems.
coco.h

roundf

```
float roundf(float x);
```

Return a floating-point integral value: floor goes toward negative infinity, trunc toward zero, and round to nearest with exact halves away from zero.
coco.h

sbyte

```
typedef signed char sbyte;
```

Eight-bit storage. Unsigned values occupy 0–255 and signed values −128–127. Cargo Count uses byte counts for crates and meals per crate.
coco.h

screen

```
void screen(byte type, byte colorset);
```

pmode records a graphics mode from 0–4 and a 512-byte-aligned buffer; it reports invalid inputs. pcls fills that buffer with a **byte pattern**. screen selects text or the prepared graphics display and color set.
coco.h · See it used in glyph.c

sdword

```
typedef signed long sdword;
```

Thirty-two-bit storage. Good for tracking absolute file lengths that exceed 65,535 bytes. The debugger shows all four bytes of the guest value.
coco.h

setBorderColor

```
byte setBorderColor(byte color);
```

Choose a six-bit hardware color for the border in CoCo 3 40- or 80-column text mode. The border and the background of a character cell are separate.
coco.h

setCaseFlag

```
void setCaseFlag(byte upperCase);
```

Select BASIC's keyboard input case flag.
coco.h

setHighSpeed

```
void setHighSpeed(BOOL fast);
```

Select the clock-speed setting. Restore the desired speed after running code whose timing assumes the normal clock.
coco.h · See it used in main.c

setOutputDevice

```
void setOutputDevice(sbyte deviceNum);
```

Change BASIC's output destination, such as screen or printer. Restore the screen when your program finishes redirecting text.

coco.h

setPmodeGraphicsAddress

```
setPmodeGraphicsAddress(newPmodeScreenBuffer)
```

Use a buffer beginning on a 512-byte boundary. The Ex form can change the drawing target while leaving a different page visible.

coco.h

setPmodeGraphicsAddressEx

```
void setPmodeGraphicsAddressEx(byte *newPmodeScreenBuffer, BOOL makeVisible);
```

Use a buffer beginning on a 512-byte boundary. The Ex form can change the drawing target while leaving a different page visible.

coco.h

setSAMRegisters

```
void setSAMRegisters(byte *samAddr, byte value, byte numBits);
```

Write a sequence of SAM control bits starting at the specified register base. Pass only values and bit counts appropriate for that hardware register.

coco.h

setTimer

```
setTimer(newValue)
```

Access the BASIC 16-bit tick count. It wraps after 65,535; setting it changes the time seen by other guest code.

coco.h

showGraphicsAddress

```
void showGraphicsAddress(byte samVRegisterValue, byte pageNum);
```

Set the displayed SAM graphics page. pageNum is a 512-byte page number, 0–127. The SAM mode value and page address must describe the same buffer layout.

coco.h

showLowResTextAddress

```
void showLowResTextAddress(void);
```

Legacy convenience entry. New programs can select a PMODE 4 buffer with pmode then screen, and return with screen(0, 0).

coco.h

showPmode4

```
void showPmode4(byte colorset);
```

Legacy convenience entry. New programs can select a PMODE 4 buffer with pmode then screen, and return with screen(0, 0).

coco.h

sinf

```
float sinf(float radians);
```

The trigonometric functions use radians. Tangent is undefined at odd multiples of π/2. The BASIC ROM computes these results in the standard CoCo configuration.

coco.h

sleep

```
void sleep(int seconds);
```

Wait for an integral number of seconds from 0 to 1092.

coco.h

sound

```
void sound(byte tone, byte duration);
```

Play a BASIC tone using its pitch and duration conventions. Color BASIC must be mapped.

coco.h

sqrtf

```
float sqrtf(float x);
```

These functions use the selected floating-point backend. For the BASIC backend keep its ROM mapped. logf requires a positive argument and sqrtf requires zero or greater; a BASIC math error can end the program.

coco.h

sword

```
typedef signed int sword;
```

Sixteen-bit storage. An unsigned word can hold 360 even though an eight-bit byte cannot. Signed values span −32,768 to 32,767.

coco.h

tanf

```
float tanf(float radians);
```

The trigonometric functions use radians. Tangent is undefined at odd multiples of π/2. The BASIC ROM computes these results in the standard CoCo configuration.

coco.h

textScreenHeight

```
extern byte textScreenHeight;
```

Legacy program state populated by initCoCoSupport. Set the required mode explicitly with width and read the mode through getTextMode for new projects.

coco.h

textScreenWidth

```
extern byte textScreenWidth;
```

Legacy program state populated by initCoCoSupport. Set the required mode explicitly with width and read the mode through getTextMode for new projects.

coco.h

TRUE

```
TRUE = 1
```

A BOOL is one byte. A zero value tests false and any nonzero value tests true. Use named constants when passing option flags.

coco.h · See it used in tone.c

truncf

```
float truncf(float x);
```

Return a floating-point integral value: floor goes toward negative infinity, trunc toward zero, and round to nearest with exact halves away from zero.
coco.h

uint16_t

```
typedef unsigned int uint16_t;
```

Sixteen-bit storage. An unsigned word can hold 360 even though an eight-bit byte cannot. Signed values span −32,768 to 32,767.
coco.h

uint32_t

```
typedef unsigned long uint32_t;
```

Thirty-two-bit storage. Good for tracking absolute file lengths that exceed 65,535 bytes. The debugger shows all four bytes of the guest value.
coco.h

uint8_t

```
typedef unsigned char uint8_t;
```

Eight-bit storage. Unsigned values occupy 0–255 and signed values −128–127. Cargo Count uses byte counts for crates and meals per crate.
coco.h

waitkey

```
byte waitkey(byte blinkCursor);
```

inkey returns zero while no character is ready. waitkey waits and accepts a cursor-blink flag. Both use the BASIC keyboard routine.
coco.h · See it used in beacon.c

width

```
byte width(byte columns);
```

Under BASIC on the CoCo 3 choose 32, 40 or 80 columns. width reports success or failure. getTextMode reports the actual width.
coco.h · See it used in planet.c

word

```
typedef unsigned int word;
```

Sixteen-bit storage. An unsigned word can hold 360 even though an eight-bit byte cannot. Signed values span −32,768 to 32,767.
coco.h · See it used in cargo.c

stdarg.h

Pass a variable-length argument list through your own function. The CoCo target supplies this header and runtime automatically.
va_arg · va_end · va_list · va_start

va_arg

```
va_arg(ap,type)
```

Initialize a cursor after the last named argument. Fetch the next argument with its actual promoted type and finish the list before leaving the function.
stdarg.h

va_end

```
va_end(ap)
```

Initialize a cursor after the last named argument. Fetch the next argument with its actual promoted type and finish the list before leaving the function.

stdarg.h

va_list

```
typedef char *va_list;
```

Initialize a cursor after the last named argument. Fetch the next argument with its actual promoted type and finish the list before leaving the function.

stdarg.h

va_start

```
va_start(ap,lastFixed)
```

Initialize a cursor after the last named argument. Fetch the next argument with its actual promoted type and finish the list before leaving the function.

stdarg.h

setjmp.h

Save and restore a nonlocal return point within a still-active call. The CoCo target supplies this header and runtime automatically.

jmp_buf · longjmp · setjmp

jmp_buf

```
typedef unsigned char jmp_buf[6];
```

Keep env alive in the calling frame. setjmp first returns zero. longjmp returns control there with a nonzero value. Jumping into a function that already returned invalidates the saved context.

setjmp.h

longjmp

```
void longjmp(jmp_buf env, int value);
```

Keep env alive in the calling frame. setjmp first returns zero. longjmp returns control there with a nonzero value. Jumping into a function that already returned invalidates the saved context.

setjmp.h

setjmp

```
int setjmp(jmp_buf env);
```

Keep env alive in the calling frame. setjmp first returns zero. longjmp returns control there with a nonzero value. Jumping into a function that already returned invalidates the saved context.

setjmp.h

assert.h

Check program assumptions at build time and while running. The CoCo target supplies this header and runtime automatically.

_FailedAssertHandler · _FreezeOnFailedAssert · _SetFailedAssertHandler · assert · static_assert

_FailedAssertHandler

```
typedef void (*_FailedAssertHandler)(const char *file, int lineno, const char *condition);
```

An enabled assertion reports its failed condition. The default handler exits; the freeze switch or a custom handler changes that action. Defining NDEBUG removes the assertion, including evaluation of its operand.
assert.h

_FreezeOnFailedAssert

```
void _FreezeOnFailedAssert(int freeze);
```

An enabled assertion reports its failed condition. The default handler exits; the freeze switch or a custom handler changes that action. Defining NDEBUG removes the assertion, including evaluation of its operand.
assert.h

_SetFailedAssertHandler

```
void _SetFailedAssertHandler(_FailedAssertHandler newHandler);
```

An enabled assertion reports its failed condition. The default handler exits; the freeze switch or a custom handler changes that action. Defining NDEBUG removes the assertion, including evaluation of its operand.
assert.h

assert

```
assert(cond)
```

An enabled assertion reports its failed condition. The default handler exits; the freeze switch or a custom handler changes that action. Defining NDEBUG removes the assertion, including evaluation of its operand.
assert.h

static_assert

```
static_assert(cond)
```

Require an integer condition at compile time. Failure prevents a build without adding target instructions.
assert.h

disk.h

Read a Disk BASIC file or address a physical floppy sector. The CoCo target supplies this header and runtime automatically.
advanceOffset · close · computeFileLength · curDriveNo · dskcon · DSKCON_READ · DSKCON_WRITE · fatBuffer · fatUpToDate · FileDesc · findDirEntry · getCurrentlyAvailableBytes · getDefaultDriveNo · getDefautlDriveNo · getFileLength · getGranuleLength · getLastBasicDriveNo · getNextSector · GRANULE_SIZE · granuleToTrack · initdisk · isLastSectorOfFile · MAX_NUM_GRANULES · normalizeFilename · openfile · read · readDiskSector · rewind · seek · setDefaultDriveNo · updateFAT · writeDiskSector

advanceOffset

```
void advanceOffset(struct FileDesc *fd, word numBytes);
```

Work within the descriptor's current cached sector. Use the available count before advancing. Refreshing the sector replaces the pointed-to data.
disk.h

close

```
sbyte close(struct FileDesc *fd);
```

Release a disk.h file descriptor after reading. Returns 0 on success or −1 on failure.
disk.h

computeFileLength

```
byte computeFileLength(uint32_t *pLength, byte firstGran, word numBytesLastSector);
```

Inspect the allocation chain or a descriptor's last sector. getFileLength places the byte length in the supplied dword on success. Distinguish that length from the rounded allocation size.
disk.h

curDriveNo

```
extern byte curDriveNo;
```

The FAT describes a drive's allocation chains. updateFAT returns the shared cache pointer or NULL on failure. The library tracks its current drive and cache freshness.
disk.h

dskcon

```
byte dskcon(byte operation, byte *buffer, byte drive, byte track, byte sector);
```

Use a full 256-byte sector buffer. Drive 0–3, track 0–34 and physical sector 1–18 refer to the mounted disk. These routines return nonzero for success. A raw write changes actual guest media.
disk.h

DSKCON_READ

```
DSKCON_READ = 2
```

Select the direction of a raw physical sector transfer. Work on a private disk image when experimenting with writes.
disk.h

DSKCON_WRITE

```
DSKCON_WRITE = 3
```

Select the direction of a raw physical sector transfer. Work on a private disk image when experimenting with writes.
disk.h

fatBuffer

```
extern byte *fatBuffer;
```

The FAT describes a drive's allocation chains. updateFAT returns the shared cache pointer or NULL on failure. The library tracks its current drive and cache freshness.
disk.h

fatUpToDate

```
extern byte fatUpToDate;
```

The FAT describes a drive's allocation chains. updateFAT returns the shared cache pointer or NULL on failure. The library tracks its current drive and cache freshness.
disk.h

FileDesc

```
struct FileDesc
```

Pass the address of a complete FileDesc to openfile. It owns the current sector cache and read position. Its size includes a whole 256-byte sector.
disk.h

findDirEntry

```
byte findDirEntry(char *dirEntry, const char *filename);
```

Search the mounted directory for a filename. Supply space for the entry data read by disk.h.
disk.h

getCurrentlyAvailableBytes

```
byte *getCurrentlyAvailableBytes(struct FileDesc *fd, word *numAvailBytes);
```

Work within the descriptor's current cached sector. Use the available count before advancing. Refreshing the sector replaces the pointed-to data.

disk.h

getDefaultDriveNo

```
getDefaultDriveNo()
```

The openfile API uses the selected drive. The misspelled getDefautlDriveNo macro is a historical alias for getDefaultDriveNo.

disk.h

getDefautlDriveNo

```
getDefautlDriveNo()
```

The openfile API uses the selected drive. The misspelled getDefautlDriveNo macro is a historical alias for getDefaultDriveNo.

disk.h

getFileLength

```
byte getFileLength(struct FileDesc *fd, uint32_t *pLength);
```

Inspect the allocation chain or a descriptor's last sector. getFileLength places the byte length in the supplied dword on success. Distinguish that length from the rounded allocation size.

disk.h

getGranuleLength

```
word getGranuleLength(byte *fat, byte granule, word numBytesLastSector);
```

Inspect the allocation chain or a descriptor's last sector. getFileLength places the byte length in the supplied dword on success. Distinguish that length from the rounded allocation size.

disk.h

getLastBasicDriveNo

```
byte getLastBasicDriveNo();
```

The openfile API uses the selected drive. The misspelled getDefautlDriveNo macro is a historical alias for getDefaultDriveNo.

disk.h

getNextSector

```
byte getNextSector(struct FileDesc *fd);
```

Work within the descriptor's current cached sector. Use the available count before advancing. Refreshing the sector replaces the pointed-to data.

disk.h

GRANULE_SIZE

```
GRANULE_SIZE = 2304
```

These values describe the library's ordinary 35-track Disk BASIC floppy. A granule spans nine 256-byte sectors; the directory has its own reserved track.

disk.h

granuleToTrack

```
void granuleToTrack(byte granule, byte *track, byte *sec);
```

Convert a granule to its track and sector offset (0 or 9). The allocation skips the directory track.

disk.h

initdisk

```
byte initdisk(byte newFATBuffer[]);
```

Provide a FAT cache of at least MAX_NUM_GRANULES bytes, kept alive while using disk.h. The return is 1 on success and 0 on failure.

disk.h

isLastSectorOfFile

```
byte isLastSectorOfFile(struct FileDesc *fd);
```

Inspect the allocation chain or a descriptor's last sector. getFileLength places the byte length in the supplied dword on success. Distinguish that length from the rounded allocation size.

disk.h

MAX_NUM_GRANULES

```
MAX_NUM_GRANULES = 68
```

These values describe the library's ordinary 35-track Disk BASIC floppy. A granule spans nine 256-byte sectors; the directory has its own reserved track.

disk.h

normalizeFilename

```
void normalizeFilename(char *dest, const char *src);
```

Convert a dotted name to an eleven-character uppercase, space-padded name plus a terminating zero. Supply twelve writable bytes.

disk.h

openfile

```
byte openfile(struct FileDesc *fd, const char *filename);
```

Prepare a real struct FileDesc with a dotted filename on the selected drive. Initialize disk.h first. A nonzero result permits reads; then close the descriptor.

disk.h

read

```
word read(struct FileDesc *fd, char *buf, word numBytesRequested);
```

Read at most the requested number of bytes into a writable buffer. Trust the returned byte count when handling a partial read.

disk.h

readDiskSector

```
byte readDiskSector(byte *dest, byte drive, byte track, byte sector);
```

Use a full 256-byte sector buffer. Drive 0–3, track 0–34 and physical sector 1–18 refer to the mounted disk. These routines return nonzero for success. A raw write changes actual guest media.

disk.h

rewind

```
void rewind(struct FileDesc *fd);
```

Reset a file's read position or choose an absolute unsigned byte offset. seek returns zero on success and −1 on failure.

disk.h

seek

```
sbyte seek(struct FileDesc *fd, uint32_t newPos);
```

Reset a file's read position or choose an absolute unsigned byte offset. seek returns zero on success and −1 on failure.

disk.h

setDefaultDriveNo

```
byte setDefaultDriveNo(byte no);
```

The openfile API uses the selected drive. The misspelled getDefautlDriveNo macro is a historical alias for getDefaultDriveNo.

disk.h

updateFAT

```
byte *updateFAT();
```

The FAT describes a drive's allocation chains. updateFAT returns the shared cache pointer or NULL on failure. The library tracks its current drive and cache freshness.

disk.h

writeDiskSector

```
byte writeDiskSector(byte *src, byte drive, byte track, byte sector);
```

Use a full 256-byte sector buffer. Drive 0–3, track 0–34 and physical sector 1–18 refer to the mounted disk. These routines return nonzero for success. A raw write changes actual guest media.

disk.h

astar.h

Find a path across a rectangular grid and walk back from the goal. Add **astar** under Project Settings → Libraries, or link with `-lastar` in your Makefile.

Save the astar example for this book

astar_advancePathIteration · astar_initNode · astar_searchGrid · astar_startPathIteration · AStarGraph · AStarNode · AStarPathIterator

astar_advancePathIteration

```
AStarNode *astar_advancePathIteration(AStarPathIterator *iter);
```

After a successful search, start at that same goal. The iterator walks **backward** toward the start. Each pointer borrows a node from the graph; NULL ends the route.

astar.h · `-lastar` · See it used in route.c

astar_initNode

```
void astar_initNode(AStarNode *node, void *userData, byte canGoEast, byte canGoSouth);
```

Initialize each tile with its optional caller-owned data and east/south exit flags. The search uses the nodes as working storage. Keep every node alive during search and path traversal.

astar.h · `-lastar` · See it used in route.c

astar_searchGrid

```
BOOL astar_searchGrid(AStarGraph *graph, size_t startX, size_t startY, size_t goalX, size_t goalY);
```

Give the graph a row-major array of width × height nodes. Coordinates must name valid tiles. A nonzero return means a route reached the goal; zero means no path to follow.

astar.h · `-lastar` · See it used in route.c

astar_startPathIteration

```
void astar_startPathIteration(AStarPathIterator *iter, AStarGraph *graph, size_t goalX, size_t goalY);
```

After a successful search, start at that same goal. The iterator walks **backward** toward the start. Each pointer borrows a node from the graph; NULL ends the route.

astar.h · `-lastar` · See it used in route.c

AStarGraph

```
typedef struct AStarGraph { … } AStarGraph;
```

Give the graph a row-major array of width × height nodes. Coordinates must name valid tiles. A nonzero return means a route reached the goal; zero means no path to follow.

astar.h · `-lastar` · See it used in route.c

AStarNode

```
typedef struct AStarNode { … } AStarNode;
```

Initialize each tile with its optional caller-owned data and east/south exit flags. The search uses the nodes as working storage. Keep every node alive during search and path traversal.

astar.h · `-lastar` · See it used in route.c

AStarPathIterator

```
typedef struct AStarPathIterator { … } AStarPathIterator;
```

After a successful search, start at that same goal. The iterator walks **backward** toward the start. Each pointer borrows a node from the graph; NULL ends the route.

astar.h · `-lastar` · See it used in route.c

Keyboard.h

Poll a guest keyboard with CoCo 3 control, function and navigation keys. Add **bcontrol** under Project Settings → Libraries, or link with `-lbcontrol` in your Makefile.

Save the bcontrol example for this book

Keyboard · Keyboard_init · Keyboard_poll · Keyboard_shutdown

Keyboard

```
typedef struct Keyboard { … } Keyboard;
```

Initialize one caller-owned keyboard object before polling. A zero result from poll means no key available. Finish with shutdown before releasing the object.

Keyboard.h · `-lbcontrol`

Keyboard_init

```
void Keyboard_init(Keyboard *keyboard);
```

Initialize one caller-owned keyboard object before polling. A zero result from poll means no key available. Finish with shutdown before releasing the object.

Keyboard.h · `-lbcontrol`

Keyboard_poll

```
byte Keyboard_poll(Keyboard *keyboard);
```

Initialize one caller-owned keyboard object before polling. A zero result from poll means no key available. Finish with shutdown before releasing the object.

Keyboard.h · `-lbcontrol`

Keyboard_shutdown

```
void Keyboard_shutdown(Keyboard *keyboard);
```

Initialize one caller-owned keyboard object before polling. A zero result from poll means no key available. Finish with shutdown before releasing the object.

Keyboard.h · `-lbcontrol`

Joystick.h

Sample analog axes and detect joystick button presses. Add **bcontrol** under Project Settings → Libraries, or link with `-lbcontrol` in your Makefile.

Save the bcontrol example for this book

Joystick_compareAxis · Joystick_disableSound · Joystick_enableSound · Joystick_getAxisValues · Joystick_getAxisValuesAndButtonState · Joystick_getButtonStates · Joystick_getButtonStates2 · Joystick_selectAnalogMuxInputs

Joystick_compareAxis

```
BOOL Joystick_compareAxis(byte referenceValue);
```

Select axis 0/1 for right X/Y or 2/3 for left X/Y. Compare against a threshold from 0–63. Nonzero means at or above threshold. Return the mux to input 0 before playing DAC sound.

Joystick.h · `-lbcontrol`

Joystick_disableSound

```
void Joystick_disableSound(void);
```

Mute the analog multiplexer around direct axis polling and restore it when finished. Otherwise the varying voltage used to measure position can be audible.

Joystick.h · `-lbcontrol` · See it used in stick.c

Joystick_enableSound

```
void Joystick_enableSound(void);
```

Mute the analog multiplexer around direct axis polling and restore it when finished. Otherwise the varying voltage used to measure position can be audible.

Joystick.h · `-lbcontrol` · See it used in stick.c

Joystick_getAxisValues

```
void Joystick_getAxisValues(byte axisValues[4], byte axisSelectionMask);
```

Supply four bytes and set mask bits for the axes to sample. 0x03 selects the right port and 0x0C the left. Results are 0–63. Disable the analog sound path while polling.

Joystick.h · `-lbcontrol` · See it used in stick.c

Joystick_getAxisValuesAndButtonState

```
byte Joystick_getAxisValuesAndButtonState(byte joystickNo, byte axisValues[2]);
```

Pass 0 for the right port or 1 for the left plus a two-byte X/Y array. 0 is left or top, 63 is right or bottom. The return is 0 for pressed and 1 for released. Surround polling with the analog sound controls.

Joystick.h · `-lbcontrol`

Joystick_getButtonStates

```
byte Joystick_getButtonStates(void);
```

The four low bits are active-low. Released buttons read as ones. The first form selects the analog input first; the second assumes the input selection is already correct.

Joystick.h · `-lbcontrol` · See it used in stick.c

Joystick_getButtonStates2

```
byte Joystick_getButtonStates2(void);
```

The four low bits are active-low. Released buttons read as ones. The first form selects the analog input first; the second assumes the input selection is already correct.

Joystick.h · `-lbcontrol`

Joystick_selectAnalogMuxInputs

```
void Joystick_selectAnalogMuxInputs(byte axis);
```

Select axis 0/1 for right X/Y or 2/3 for left X/Y. Compare against a threshold from 0–63. Nonzero means at or above threshold. Return the mux to input 0 before playing DAC sound.

Joystick.h · `-lbcontrol` · See it used in stick.c

catmull-rom.h

Interpolate a smooth path through a sequence of points. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

CatmullRom_drawLine · CatmullRom_onPieceEnd · drawCatmullRomPath

CatmullRom_drawLine

```
typedef BOOL (*CatmullRom_drawLine)(short x0, short y0, short x1, short y1, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

catmull-rom.h · `-lbgraph`

CatmullRom_onPieceEnd

```
typedef BOOL (*CatmullRom_onPieceEnd)(size_t pointIndex, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

catmull-rom.h · `-lbgraph`

drawCatmullRomPath

```
BOOL drawCatmullRomPath(BOOL openPath, const ShortPoint *points, size_t numPoints, byte
numSegmentsPerPiece, CatmullRom_onPieceEnd onPieceEnd, CatmullRom_drawLine drawLine, void *cookie);
```

Supply at least four points and at least two segments per piece. Keep the line callback and its cookie valid. A callback returning false stops the drawing and produces a false result.

catmull-rom.h · `-lbgraph`

clipping.h

Clip a line to a rectangle before drawing it. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

cohenSutherlandLineClip · ShortClipRect · ShortPoint

cohenSutherlandLineClip

```
BOOL cohenSutherlandLineClip(short *x0, short *y0, short *x1, short *y1, const ShortClipRect
*rect);
```

Clip the given segment to an ordered rectangle, with y increasing downward. A true result gives clipped endpoints through the four pointers; false rejects the segment.

clipping.h · `-lbgraph`

ShortClipRect

```
typedef struct ShortClipRect { … } ShortClipRect;
```

Signed pixel coordinates for the clipping routine. Supply a top-left corner and a bottom-right corner, with increasing y toward the bottom.

clipping.h · `-lbgraph`

ShortPoint

```
typedef struct ShortPoint { … } ShortPoint;
```

Signed pixel coordinates for the clipping routine. Supply a top-left corner and a bottom-right corner, with increasing y toward the bottom.

clipping.h · `-lbgraph`

Drawer.h

Move a drawing pen and run a DRAW script in a graphics buffer. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

Drawer · Drawer_draw · Drawer_getPosition · Drawer_init · Drawer_initForHSCREEN · Drawer_initForPMODE · Drawer_setColorIndex · Drawer_setPosition

Drawer

```
typedef struct Drawer { … } Drawer;
```

Keep a writable screen buffer and a Drawer alive together. Supply dimensions and pixels per byte, or let the mode-specific initializer derive them. A false return rejects an invalid setup. Initialization positions the pen in the center; displaying the buffer is a separate operation.

Drawer.h · `-lbgraph`

Drawer_draw

```
const char *Drawer_draw(Drawer *d, const char *script);
```

Interpret DRAW commands against the initialized buffer. Coordinates must stay inside its actual pixel dimensions. NULL means the whole script was accepted; an error pointer identifies the problem character in the original script.

Drawer.h · `-lbgraph`

Drawer_getPosition

```
void Drawer_getPosition(Drawer *d, word *pX, byte *pY);
```

Set a palette index valid for the configured depth. Positions are pixel coordinates relative to the buffer. The getter writes both coordinates through caller-supplied pointers.

Drawer.h · `-lbgraph`

Drawer_init

```
BOOL Drawer_init(Drawer *d, byte *screenBuffer, word screenWidthInPixels, byte
screenHeightInPixels, byte pixelsPerByte);
```

Keep a writable screen buffer and a Drawer alive together. Supply dimensions and pixels per byte, or let the mode-specific initializer derive them. A false return rejects an invalid setup. Initialization positions the pen in the center; displaying the buffer is a separate operation.

Drawer.h · `-lbgraph`

Drawer_initForHSCREEN

```
BOOL Drawer_initForHSCREEN(Drawer *d, byte *screenBuffer, byte modeNo);
```

Keep a writable screen buffer and a Drawer alive together. Supply dimensions and pixels per byte, or let the mode-specific initializer derive them. A false return rejects an invalid setup. Initialization positions the pen in the center; displaying the buffer is a separate operation.
Drawer.h · `-lbgraph`

Drawer_initForPMODE

```
BOOL Drawer_initForPMODE(Drawer *d, byte *screenBuffer, byte pmodeNo);
```

Keep a writable screen buffer and a Drawer alive together. Supply dimensions and pixels per byte, or let the mode-specific initializer derive them. A false return rejects an invalid setup. Initialization positions the pen in the center; displaying the buffer is a separate operation.
Drawer.h · `-lbgraph`

Drawer_setColorIndex

```
void Drawer_setColorIndex(Drawer *d, byte newColorIndex);
```

Set a palette index valid for the configured depth. Positions are pixel coordinates relative to the buffer. The getter writes both coordinates through caller-supplied pointers.
Drawer.h · `-lbgraph`

Drawer_setPosition

```
void Drawer_setPosition(Drawer *d, word x, byte y);
```

Set a palette index valid for the configured depth. Positions are pixel coordinates relative to the buffer. The getter writes both coordinates through caller-supplied pointers.
Drawer.h · `-lbgraph`

lores.h

Read or change the color of a low-resolution semigraphics block. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.
Save the bgraph example for this book
point · reset · set

point

```
char point(const byte *screen, byte x, byte y);
```

Use the 32 × 16 character buffer as a 64 × 32 semigraphics grid. Coordinates begin at zero; set accepts colors 1–8. point returns the color or −1 for a text character.
lores.h · `-lbgraph`

reset

```
void reset(byte *screen, byte x, byte y);
```

Use the 32 × 16 character buffer as a 64 × 32 semigraphics grid. Coordinates begin at zero; set accepts colors 1–8. point returns the color or −1 for a text character.
lores.h · `-lbgraph`

set

```
void set(byte *screen, byte x, byte y, byte color);
```

Use the 32 × 16 character buffer as a 64 × 32 semigraphics grid. Coordinates begin at zero; set accepts colors 1–8. point returns the color or −1 for a text character.
lores.h · `-lbgraph`

Painter.h

Flood-fill a connected area with caller-owned scratch space. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

Painter · Painter_continuationFuncPtr · Painter_getPixelFuncPtr · Painter_init · Painter_INVALID_ARGUMENT · Painter_NO_ERROR · Painter_OUT_OF_STACK · Painter_paint · Painter_setPixelFuncPtr

Painter

```
typedef struct Painter { … } Painter;
```

Prepare a flood fill with pixel callbacks, actual pixel dimensions and a caller-owned workspace kept alive during painting. A 64-byte workspace is a useful starting point. False indicates invalid setup.

Painter.h · `-lbgraph`

Painter_continuationFuncPtr

```
typedef BOOL (*Painter_continuationFuncPtr)(void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Painter.h · `-lbgraph`

Painter_getPixelFuncPtr

```
typedef byte (*Painter_getPixelFuncPtr)(word x, byte y, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Painter.h · `-lbgraph`

Painter_init

```
BOOL Painter_init(Painter *p, Painter_getPixelFuncPtr getPixel, Painter_setPixelFuncPtr setPixel,
Painter_continuationFuncPtr continuationFuncPtr, word screenWidthInPixels, byte
screenHeightInPixels, byte *workSpace, size_t workSpaceCapacity);
```

Prepare a flood fill with pixel callbacks, actual pixel dimensions and a caller-owned workspace kept alive during painting. A 64-byte workspace is a useful starting point. False indicates invalid setup.

Painter.h · `-lbgraph`

Painter_INVALID_ARGUMENT

```
Painter_INVALID_ARGUMENT = 1
```

A Painter_paint status. Check invalid coordinates or colors for an argument failure and increase the **Painter workspace** for an out-of-stack result.

Painter.h · `-lbgraph`

Painter_NO_ERROR

```
Painter_NO_ERROR = 0
```

A Painter_paint status. Check invalid coordinates or colors for an argument failure and increase the **Painter workspace** for an out-of-stack result.

Painter.h · `-lbgraph`

Painter_OUT_OF_STACK

```
Painter_OUT_OF_STACK = 2
```

A Painter_paint status. Check invalid coordinates or colors for an argument failure and increase the **Painter workspace** for an out-of-stack result.

Painter.h · `-lbgraph`

Painter_paint

```
byte Painter_paint(Painter *p, word x, byte y, byte newColor, byte oldColor, void *cookie);
```

Fill from (x,y) using the configured callbacks and their cookie. Give distinct old and new color indices. Painter_OUT_OF_STACK means this shape exceeded the supplied scratch buffer.

Painter.h · `-lbgraph`

Painter_setPixelFuncPtr

```
typedef void (*Painter_setPixelFuncPtr)(word x, byte y, byte color, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Painter.h · `-lbgraph`

Primitive.h

Plot one pixel or draw lines and shapes using a buffer-specific callback. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

Primitive_15DegreeCircleArcs · Primitive_circle · Primitive_ellipse · Primitive_filledRectangle · Primitive_getAddressAndMaskPmode3 · Primitive_getAddressAndMaskPmode4 · Primitive_getPixel256x16 · Primitive_getPixel320x16 · Primitive_getPixelFuncPtr · Primitive_getPixelPmode3 · Primitive_getPixelPmode4 · Primitive_line · Primitive_lineFuncPtr · Primitive_rectangle · Primitive_setPixel256x16 · Primitive_setPixel320x16 · Primitive_setPixelFuncPtr · Primitive_setPixelPmode3 · Primitive_setPixelPmode4

Primitive_15DegreeCircleArcs

```
void Primitive_15DegreeCircleArcs(word x, byte y, word radius, byte arcBegin, byte arcEnd, byte
color, Primitive_lineFuncPtr drawLine, Primitive_setPixelFuncPtr setPixel, void *cookie);
```

Use a pixel callback for the current buffer and keep the whole shape in bounds. Ellipse scale factors use 256 for one whole unit; the arc routine measures angles in 15-degree steps.

Primitive.h · `-lbgraph`

Primitive_circle

```
Primitive_circle(x,y,radius,color,setPixel,cookie)
```

Use a pixel callback for the current buffer and keep the whole shape in bounds. Ellipse scale factors use 256 for one whole unit; the arc routine measures angles in 15-degree steps.

Primitive.h · `-lbgraph`

Primitive_ellipse

```
void Primitive_ellipse(word x, byte y, word radius, byte color, word horizontalFactor, word
verticalFactor, Primitive_setPixelFuncPtr setPixel, void *cookie);
```

Use a pixel callback for the current buffer and keep the whole shape in bounds. Ellipse scale factors use 256 for one whole unit; the arc routine measures angles in 15-degree steps.

Primitive.h · `-lbgraph`

Primitive_filledRectangle

```
void Primitive_filledRectangle(word x0, byte y0, word x1, byte y1, byte color,
Primitive_setPixelFuncPtr setPixel, void *cookie);
```

Draw between inclusive pixel coordinates using the supplied pixel callback. Pass its required buffer or state as cookie. Clip coordinates to that buffer before drawing.

Primitive.h · `-lbgraph`

Primitive_getAddressAndMaskPmode3

```
void Primitive_getAddressAndMaskPmode3(word x, byte y, byte color, void *pmode3Buffer, byte **bytePtr, byte *andMask, byte *orMask);
```

A PMODE 3 buffer holds 128 × 192 two-bit pixels in 6,144 bytes. Use x 0–127, y 0–191 and a color of 0–3. The mask helper gives a byte address and the single-pixel masks.

Primitive.h · `-lbgraph`

Primitive_getAddressAndMaskPmode4

```
void Primitive_getAddressAndMaskPmode4(word x, byte y, byte color, void *pmode4Buffer, byte **bytePtr, byte *andMask, byte *orMask);
```

A PMODE 4 buffer holds 256 × 192 one-bit pixels in 6,144 bytes. Use x 0–255, y 0–191 and a color of 0 or 1. The mask helper returns an address and masks for changing one pixel while preserving its neighbors.

Primitive.h · `-lbgraph`

Primitive_getPixel256x16

```
byte Primitive_getPixel256x16(word x, byte y, void *screenBuffer);
```

Work in a 256-wide, four-bit pixel buffer. Two neighboring pixels share a byte, so each row takes 128 bytes. Supply x 0–255, y within the buffer height and a color index 0–15.

Primitive.h · `-lbgraph`

Primitive_getPixel320x16

```
byte Primitive_getPixel320x16(word x, byte y, void *screenBuffer);
```

Work in a 320-wide, four-bit pixel buffer. Its stride is **160 bytes** per row. Use x 0–319, y within the allocated height and a color index 0–15. These calls access the supplied buffer whether or not that buffer is currently visible.

Primitive.h · `-lbgraph` · See it used in beacon.c

Primitive_getPixelFuncPtr

```
typedef byte (*Primitive_getPixelFuncPtr)(word x, byte y, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Primitive.h · `-lbgraph`

Primitive_getPixelPmode3

```
byte Primitive_getPixelPmode3(word x, byte y, void *pmode3Buffer);
```

A PMODE 3 buffer holds 128 × 192 two-bit pixels in 6,144 bytes. Use x 0–127, y 0–191 and a color of 0–3. The mask helper gives a byte address and the single-pixel masks.

Primitive.h · `-lbgraph`

Primitive_getPixelPmode4

```
byte Primitive_getPixelPmode4(word x, byte y, void *pmode4Buffer);
```

A PMODE 4 buffer holds 256 × 192 one-bit pixels in 6,144 bytes. Use x 0–255, y 0–191 and a color of 0 or 1. The mask helper returns an address and masks for changing one pixel while preserving its neighbors.

Primitive.h · `-lbgraph`

Primitive_line

```
void Primitive_line(word x0, byte y0, word x1, byte y1, byte color, Primitive_setPixelFuncPtr setPixel, void *cookie);
```

Draw between inclusive pixel coordinates using the supplied pixel callback. Pass its required buffer or state as cookie. Clip coordinates to that buffer before drawing.

Primitive.h · `-lbgraph`

Primitive_lineFuncPtr

```
typedef void (*Primitive_lineFuncPtr)(word x0, byte y0, word x1, byte y1, byte color,
Primitive_setPixelFuncPtr setPixel, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Primitive.h · `-lbgraph`

Primitive_rectangle

```
void Primitive_rectangle(word x0, byte y0, word x1, byte y1, byte color, Primitive_setPixelFuncPtr
setPixel, void *cookie);
```

Draw between inclusive pixel coordinates using the supplied pixel callback. Pass its required buffer or state as cookie. Clip coordinates to that buffer before drawing.

Primitive.h · `-lbgraph`

Primitive_setPixel256x16

```
void Primitive_setPixel256x16(word x, byte y, byte color, void *screenBuffer);
```

Work in a 256-wide, four-bit pixel buffer. Two neighboring pixels share a byte, so each row takes 128 bytes. Supply x 0–255, y within the buffer height and a color index 0–15.

Primitive.h · `-lbgraph`

Primitive_setPixel320x16

```
void Primitive_setPixel320x16(word x, byte y, byte color, void *screenBuffer);
```

Work in a 320-wide, four-bit pixel buffer. Its stride is **160 bytes** per row. Use x 0–319, y within the allocated height and a color index 0–15. These calls access the supplied buffer whether or not that buffer is currently visible.

Primitive.h · `-lbgraph` · See it used in beacon.c

Primitive_setPixelFuncPtr

```
typedef void (*Primitive_setPixelFuncPtr)(word x, byte y, byte color, void *cookie);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Primitive.h · `-lbgraph`

Primitive_setPixelPmode3

```
void Primitive_setPixelPmode3(word x, byte y, byte color, void *pmode3Buffer);
```

A PMODE 3 buffer holds 128 × 192 two-bit pixels in 6,144 bytes. Use x 0–127, y 0–191 and a color of 0–3. The mask helper gives a byte address and the single-pixel masks.

Primitive.h · `-lbgraph`

Primitive_setPixelPmode4

```
void Primitive_setPixelPmode4(word x, byte y, byte color, void *pmode4Buffer);
```

A PMODE 4 buffer holds 256 × 192 one-bit pixels in 6,144 bytes. Use x 0–255, y 0–191 and a color of 0 or 1. The mask helper returns an address and masks for changing one pixel while preserving its neighbors.

Primitive.h · `-lbgraph`

Trig.h

Use 15-degree sine and cosine increments without floating point. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

compute15DegreeCosSin · cosSin15DegreeTable · CosSinFactorCouple

compute15DegreeCosSin

```
void compute15DegreeCosSin(byte i, BOOL *cosNegated, word *cosAbsValue, BOOL *sinNegated, word *sinAbsValue);
```

Express the angle in 15-degree steps: index 0 is 0°, 6 is 90° and 24 is 360°. The helper returns a separate sign and unsigned magnitude scaled to 65,535 for each result.

Trig.h · `-lbgraph`

cosSin15DegreeTable

```
extern const CosSinFactorCouple cosSin15DegreeTable[4];
```

Express the angle in 15-degree steps: index 0 is 0°, 6 is 90° and 24 is 360°. The helper returns a separate sign and unsigned magnitude scaled to 65,535 for each result.

Trig.h · `-lbgraph`

CosSinFactorCouple

```
typedef struct CosSinFactorCouple { … } CosSinFactorCouple;
```

One pair of unsigned sine/cosine magnitudes in the 15-degree lookup table. Use compute15DegreeCosSin to obtain the quadrant signs.

Trig.h · `-lbgraph`

semigr.h

Manipulate legacy semigraphics pixel buffers. Add **bgraph** under Project Settings → Libraries, or link with `-lbgraph` in your Makefile.

Save the bgraph example for this book

GetPixelFunction · getPixelSG12 · getPixelSG24 · getPixelSG8 · SetPixelFunction · setPixelSG12 · setPixelSG24 · setPixelSG8 · sgMode

GetPixelFunction

```
typedef byte (*GetPixelFunction)(byte *screenBuffer, byte x, byte y);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

semigr.h · `-lbgraph`

getPixelSG12

```
byte getPixelSG12(byte *screenBuffer, byte x, byte y);
```

Use an SG12 buffer. The grid spans x 0–63 and y 0–95. Changing buffer bytes and observing that mode on the hardware are separate checks.

semigr.h · `-lbgraph`

getPixelSG24

```
byte getPixelSG24(byte *screenBuffer, byte x, byte y);
```

Use an SG24 buffer. The grid spans x 0–63 and y 0–191. Match the buffer format and color convention before drawing.

semigr.h · `-lbgraph`

getPixelSG8

```
byte getPixelSG8(byte *screenBuffer, byte x, byte y);
```

Use an SG8 buffer. The grid spans x 0–63 and y 0–63. The returned byte is the pixel's color index.

semigr.h · `-lbgraph`

SetPixelFunction

```
typedef void (*SetPixelFunction)(byte *screenBuffer, byte x, byte y, byte color);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

semigr.h · `-lbgraph`

setPixelSG12

```
void setPixelSG12(byte *screenBuffer, byte x, byte y, byte color);
```

Use an SG12 buffer. The grid spans x 0–63 and y 0–95. Changing buffer bytes and observing that mode on the hardware are separate checks.

semigr.h · `-lbgraph`

setPixelSG24

```
void setPixelSG24(byte *screenBuffer, byte x, byte y, byte color);
```

Use an SG24 buffer. The grid spans x 0–63 and y 0–191. Match the buffer format and color convention before drawing.

semigr.h · `-lbgraph`

setPixelSG8

```
void setPixelSG8(byte *screenBuffer, byte x, byte y, byte color);
```

Use an SG8 buffer. The grid spans x 0–63 and y 0–63. The returned byte is the pixel's color index.

semigr.h · `-lbgraph`

sgMode

```
BOOL sgMode(byte mode, void *screenBuffer);
```

Select a semigraphics format and its buffer. Pass mode 8, 12 or 24, or 0 for ordinary SG4. Check its Boolean result and the display's hardware support.

semigr.h · `-lbgraph`

Player.h

Generate music with the CoCo sound DAC and interrupt timer. Add **bsound** under Project Settings → Libraries, or link with `-lbsound` in your Makefile.

Save the bsound example for this book

Player · Player_continuationFuncPtr · PLAYER_DAC_SAMPLE_VALUE_SHIFT · Player_defaultIRQService · Player_disableSound · Player_enableSound · Player_getSamplePlayDelayLength · Player_init · PLAYER_MAX_AMPLITUDE · PLAYER_MAX_LEVEL · PLAYER_MID_LEVEL · PLAYER_MIN_LEVEL · Player_play · Player_playSample · Player_shutdown · Player_sound

Player

```
typedef struct Player { … } Player;
```

Keep one Player alive throughout playback. Pass true for library-owned 60 Hz IRQ service, or false and supply one tick yourself every sixtieth of a second. Choose one clock source. Shutdown removes the library's hook when installed and disables its analog output.

Player.h · `-lbsound` · See it used in tone.c

Player_continuationFuncPtr

```
typedef BOOL (*Player_continuationFuncPtr)(void);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Player.h · `-lbsound`

PLAYER_DAC_SAMPLE_VALUE_SHIFT

`PLAYER_DAC_SAMPLE_VALUE_SHIFT = 2`

The guest DAC accepts six significant bits. Player samples store them in the upper six bits of a byte. Use the shift constant when turning levels into sample bytes.

Player.h · `-lbsound`

Player_defaultIRQService

```
void Player_defaultIRQService(void);
```

Keep one Player alive throughout playback. Pass true for library-owned 60 Hz IRQ service, or false and supply one tick yourself every sixtieth of a second. Choose one clock source. Shutdown removes the library's hook when installed and disables its analog output.

Player.h · `-lbsound`

Player_disableSound

```
void Player_disableSound(void);
```

Open or close the analog path to the CoCo sound output. This shares hardware with joystick measurements.

Player.h · `-lbsound`

Player_enableSound

```
void Player_enableSound(void);
```

Open or close the analog path to the CoCo sound output. This shares hardware with joystick measurements.

Player.h · `-lbsound`

Player_getSamplePlayDelayLength

```
word Player_getSamplePlayDelayLength(unsigned long cpuCyclesPerSecond, word playBackFrequencyInHz);
```

Provide a nonempty sample buffer and a pointer one byte past the last sample. Each byte uses its upper six bits for the DAC. Use CPU frequency and sample rate in Hz to compute the delay. Sample playback expects the 60 Hz IRQ disabled, unlike Player_sound.

Player.h · `-lbsound`

Player_init

```
void Player_init(Player *player, BOOL useInternalIRQService, Player_continuationFuncPtr
contFuncPtr);
```

Keep one Player alive throughout playback. Pass true for library-owned 60 Hz IRQ service, or false and supply one tick yourself every sixtieth of a second. Choose one clock source. Shutdown removes the library's hook when installed and disables its analog output.

Player.h · `-lbsound` · See it used in tone.c

PLAYER_MAX_AMPLITUDE

`PLAYER_MAX_AMPLITUDE = 31`

The guest DAC accepts six significant bits. Player samples store them in the upper six bits of a byte. Use the shift constant when turning levels into sample bytes.

Player.h · `-lbsound`

PLAYER_MAX_LEVEL

`PLAYER_MAX_LEVEL = 63`

The guest DAC accepts six significant bits. Player samples store them in the upper six bits of a byte. Use the shift constant when turning levels into sample bytes.

Player.h · `-lbsound`

PLAYER_MID_LEVEL

`PLAYER_MID_LEVEL = 31`

The guest DAC accepts six significant bits. Player samples store them in the upper six bits of a byte. Use the shift constant when turning levels into sample bytes.

Player.h · `-lbsound`

PLAYER_MIN_LEVEL

`PLAYER_MIN_LEVEL = 0`

The guest DAC accepts six significant bits. Player samples store them in the upper six bits of a byte. Use the shift constant when turning levels into sample bytes.

Player.h · `-lbsound`

Player_play

```
const char *Player_play(const char *script);
```

Play an Extended BASIC-style note string with interrupts ticking. NULL means the script finished or your continuation callback requested a stop. Any other return points to the rejected character. X substring commands and variable substitution are unsupported.

Player.h · `-lbsound`

Player_playSample

```
void Player_playSample(const void *sampleArray, const void *sampleArrayEnd, word delayLength);
```

Provide a nonempty sample buffer and a pointer one byte past the last sample. Each byte uses its upper six bits for the DAC. Use CPU frequency and sample rate in Hz to compute the delay. Sample playback expects the 60 Hz IRQ disabled, unlike Player_sound.

Player.h · `-lbsound`

Player_shutdown

```
void Player_shutdown(void);
```

Keep one Player alive throughout playback. Pass true for library-owned 60 Hz IRQ service, or false and supply one tick yourself every sixtieth of a second. Choose one clock source. Shutdown removes the library's hook when installed and disables its analog output.

Player.h · `-lbsound` · See it used in tone.c

Player_sound

```
void Player_sound(byte tone, byte duration);
```

With the player initialized and its 60 Hz IRQ running, play a tone and duration from 1–255. Invalid arguments produce no tone.

Player.h · `-lbsound` · See it used in tone.c

SquareWaveGenerator.h

Produce square-wave samples for a chosen amplitude and period. Add **bsound** under Project Settings → Libraries, or link with `-lbsound` in your Makefile.

Save the bsound example for this book

SquareWaveGenerator · SquareWaveGenerator_getNextSamples · SquareWaveGenerator_init · SquareWaveGenerator_setNextNote

SquareWaveGenerator

```
typedef struct SquareWaveGenerator { … } SquareWaveGenerator;
```

The guest DAC accepts six significant bits. Player samples store them in the upper six bits of a byte. Use the shift constant when turning levels into sample bytes.

SquareWaveGenerator.h · `-lbsound`

SquareWaveGenerator_getNextSamples

```
void SquareWaveGenerator_getNextSamples(SquareWaveGenerator *self, byte *begin, byte *end);
```

Prepare a square wave using an amplitude and half-period measured in samples. Write into the nonempty caller-supplied range up to but excluding end. A pending note change takes effect at the period boundary.

SquareWaveGenerator.h · `-lbsound`

SquareWaveGenerator_init

```
void SquareWaveGenerator_init(SquareWaveGenerator *self, byte diffFromMidLevel, size_t
numSamplesInHalfPeriod);
```

Prepare a square wave using an amplitude and half-period measured in samples. Write into the nonempty caller-supplied range up to but excluding end. A pending note change takes effect at the period boundary.

SquareWaveGenerator.h · `-lbsound`

SquareWaveGenerator_setNextNote

```
void SquareWaveGenerator_setNextNote(SquareWaveGenerator *self, size_t numSamplesInHalfPeriod);
```

Prepare a square wave using an amplitude and half-period measured in samples. Write into the nonempty caller-supplied range up to but excluding end. A pending note change takes effect at the period boundary.

SquareWaveGenerator.h · `-lbsound`

cardgame.h

Build a deck, select a card and render game graphics. Add **cardgame** under Project Settings → Libraries, or link with `-lcardgame` in your Makefile.

Save the cardgame example for this book

BAD_CARD_SUIT · BAD_CARD_VALUE · BAD_INDEX · beep · BEEP_MAX_VOL · BEEP_MIN_VOL · BYTES_PER_CARD_WIDTH · closeCardGame · drawCompiledCard · drawCompiledCard2 · drawCompiledPixMap · drawFaceDownCard · drawSuit · eraseCard · getCardType · getCardValue · getNumPixelRowsPerCard · getSuitName · getTicksForFrequency · initSound · NUMCARDS · openCardGame · pickCardFromDeck · ROWS_ABOVE_SUIT · ROWS_BELOW_SUIT · ROWS_PER_CARD_BOTTOM · ROWS_PER_CARD_TOP · ROWS_PER_CARD_VALUE · ROWS_PER_SUIT · shuffleDeck · SUIT_CLUBS · SUIT_DIAMONDS · SUIT_HEARTS · SUIT_JOKER · SUIT_NONE · SUIT_SPADES · whiteNoise

BAD_CARD_SUIT

```
BAD_CARD_SUIT = 0xFF
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.

cardgame.h · `-lcardgame`

BAD_CARD_VALUE

```
BAD_CARD_VALUE = 0
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.

cardgame.h · `-lcardgame`

BAD_INDEX

```
BAD_INDEX = 0xFF
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.

cardgame.h · `-lcardgame`

beep

```
void beep(byte duration, word ticks, byte volume);
```

Initialize this card library's sound support before use. getTicksForFrequency converts Hz to the period value used by beep. Keep the period positive and the volume within BEEP_MAX_VOL.

cardgame.h · `-lcardgame`

BEEP_MAX_VOL

```
BEEP_MAX_VOL = 63
```

Volume limits for CardGame's beep and whiteNoise. Initialize its sound hardware first.

cardgame.h · `-lcardgame`

BEEP_MIN_VOL

```
BEEP_MIN_VOL = 0
```

Volume limits for CardGame's beep and whiteNoise. Initialize its sound hardware first.

cardgame.h · `-lcardgame`

BYTES_PER_CARD_WIDTH

```
BYTES_PER_CARD_WIDTH = 4
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.

cardgame.h · `-lcardgame`

closeCardGame

```
void closeCardGame(void);
```

Reserve the aligned 6,144-byte PMODE 4 buffer before opening a graphics game. The open/close pairs own their screen and console state while active. Use screen(0,0) when returning to ordinary 32-column text.

cardgame.h · `-lcardgame`

drawCompiledCard

```
void drawCompiledCard(byte cardValue, byte cardSuit, byte byteColumn, byte pixelRow);
```

Work on the prepared game buffer. Horizontal card columns are **byte offsets** (0–28 for a full card), while vertical position is in pixel rows. A 32-pixel-wide card uses four bytes per row.

cardgame.h · `-lcardgame`

drawCompiledCard2

```
void drawCompiledCard2(byte cardValue, byte cardSuit, byte byteColumn, byte pixelRow, char
suitRowOffset, byte maxNumRowsToDraw);
```

Work on the prepared game buffer. Horizontal card columns are **byte offsets** (0–28 for a full card), while vertical position is in pixel rows. A 32-pixel-wide card uses four bytes per row.

cardgame.h · `-lcardgame`

drawCompiledPixMap

```
byte *drawCompiledPixMap(byte byteOffset, byte pixelRow, const word *wordArray, byte numRows, byte
rowRepetitions);
```

A compiled picture supplies one 16-bit row per input element. Both source rows and repetition count must be positive. Use the height query when planning the card's on-screen bounds.

cardgame.h · `-lcardgame`

drawFaceDownCard

```
void drawFaceDownCard(byte rowInPixels, byte colInBytes);
```

Work on the prepared game buffer. Horizontal card columns are **byte offsets** (0–28 for a full card), while vertical position is in pixel rows. A 32-pixel-wide card uses four bytes per row.
cardgame.h · `-lcardgame`

drawSuit

```
void drawSuit(byte col, byte row, byte suit);
```

Work on the prepared game buffer. Horizontal card columns are **byte offsets** (0–28 for a full card), while vertical position is in pixel rows. A 32-pixel-wide card uses four bytes per row.
cardgame.h · `-lcardgame`

eraseCard

```
void eraseCard(byte byteColumn, byte pixelRow);
```

Work on the prepared game buffer. Horizontal card columns are **byte offsets** (0–28 for a full card), while vertical position is in pixel rows. A 32-pixel-wide card uses four bytes per row.
cardgame.h · `-lcardgame`

getCardType

```
byte getCardType(byte card);
```

Decode a card to a suit and value. Ordinary ranks are 1–13; jokers use their own suit and values 1–2. Check for BAD_CARD_SUIT or BAD_CARD_VALUE on unexpected input. Suit names are borrowed strings.
cardgame.h · `-lcardgame`

getCardValue

```
byte getCardValue(byte card);
```

Decode a card to a suit and value. Ordinary ranks are 1–13; jokers use their own suit and values 1–2. Check for BAD_CARD_SUIT or BAD_CARD_VALUE on unexpected input. Suit names are borrowed strings.
cardgame.h · `-lcardgame`

getNumPixelRowsPerCard

```
byte getNumPixelRowsPerCard();
```

A compiled picture supplies one 16-bit row per input element. Both source rows and repetition count must be positive. Use the height query when planning the card's on-screen bounds.
cardgame.h · `-lcardgame`

getSuitName

```
const char *getSuitName(byte suit);
```

Decode a card to a suit and value. Ordinary ranks are 1–13; jokers use their own suit and values 1–2. Check for BAD_CARD_SUIT or BAD_CARD_VALUE on unexpected input. Suit names are borrowed strings.
cardgame.h · `-lcardgame`

getTicksForFrequency

```
word getTicksForFrequency(word hertz);
```

Initialize this card library's sound support before use. getTicksForFrequency converts Hz to the period value used by beep. Keep the period positive and the volume within BEEP_MAX_VOL.
cardgame.h · `-lcardgame`

initSound

```
void initSound();
```

Initialize this card library's sound support before use. getTicksForFrequency converts Hz to the period value used by beep. Keep the period positive and the volume within BEEP_MAX_VOL.
cardgame.h · `-lcardgame`

NUMCARDS

```
NUMCARDS = 54
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.

cardgame.h · `-lcardgame`

openCardGame

```
void openCardGame(void *_scrnBuffer, byte _emulatePMode4OnCoCo3);
```

Reserve the aligned 6,144-byte PMODE 4 buffer before opening a graphics game. The open/close pairs own their screen and console state while active. Use screen(0,0) when returning to ordinary 32-column text.

cardgame.h · `-lcardgame`

pickCardFromDeck

```
byte pickCardFromDeck(byte deck[], byte *pNumCards);
```

Keep a mutable deck array and its current count. Shuffle the chosen number of cards, then pass the address of the count to deal one. Check that a card remains before dealing.

cardgame.h · `-lcardgame` · See it used in deal.c

ROWS_ABOVE_SUIT

```
ROWS_ABOVE_SUIT = 2
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.

cardgame.h · `-lcardgame`

ROWS_BELOW_SUIT

```
ROWS_BELOW_SUIT = 6
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.

cardgame.h · `-lcardgame`

ROWS_PER_CARD_BOTTOM

```
ROWS_PER_CARD_BOTTOM = ROWS_PER_CARD_TOP
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.

cardgame.h · `-lcardgame`

ROWS_PER_CARD_TOP

```
ROWS_PER_CARD_TOP = 5
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.

cardgame.h · `-lcardgame`

ROWS_PER_CARD_VALUE

```
ROWS_PER_CARD_VALUE = 10
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.

cardgame.h · `-lcardgame`

ROWS_PER_SUIT

```
ROWS_PER_SUIT = 7
```

Layout of CardGame's compiled artwork. Vertical measurements are pixel rows; horizontal width counts bytes in the PMODE 4 buffer.
cardgame.h · `-lcardgame`

shuffleDeck

```
void shuffleDeck(byte deck[], byte numCards);
```

Keep a mutable deck array and its current count. Shuffle the chosen number of cards, then pass the address of the count to deal one. Check that a card remains before dealing.
cardgame.h · `-lcardgame`

SUIT_CLUBS

```
SUIT_CLUBS = 3
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.
cardgame.h · `-lcardgame`

SUIT_DIAMONDS

```
SUIT_DIAMONDS = 2
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.
cardgame.h · `-lcardgame`

SUIT_HEARTS

```
SUIT_HEARTS = 0
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.
cardgame.h · `-lcardgame`

SUIT_JOKER

```
SUIT_JOKER = 4
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.
cardgame.h · `-lcardgame`

SUIT_NONE

```
SUIT_NONE = 255
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.
cardgame.h · `-lcardgame`

SUIT_SPADES

```
SUIT_SPADES = 1
```

CardGame's deck includes the usual four suits and two jokers. The named bad values represent an invalid suit, rank or index; do not use them to index the drawing tables.
cardgame.h · `-lcardgame`

whiteNoise

```
void whiteNoise(word duration, word spacing, byte volume);
```

Initialize this card library's sound support before use. getTicksForFrequency converts Hz to the period value used by beep. Keep the period positive and the volume within BEEP_MAX_VOL.
cardgame.h · `-lcardgame`

pmode4game.h

Set up the game buffer, graphics cursor, text and keyboard interaction. Add **cardgame** under Project Settings → Libraries, or link with `-lcardgame` in your Makefile.

Save the cardgame example for this book

assertf · BYTES_PER_SCREEN · BYTES_PER_SCREEN_ROW · BYTES_PER_TEXT_ROW · closePMode4Game · concat · drawChar · drawCharAtAddressWithoutMovingCursor · drawCharWithoutMovingCursor · emulatingPMode4OnCoCo3 · enableInvertedColorText · getCurrentCharPos · getCursor · GetTimerFunction · hiResTextConsoleOutHook · invertCharAtCursor · invertCharAtCursorExceptRowBelowGlyph · KeyboardPollFunction · makeSpacesTransparent · moveCursor · moveCursorLeft · oldCHROUT · openPMode4Game · optDelay · PIXEL_ROWS_PER_GLYPH · PIXEL_ROWS_PER_SCREEN · PIXEL_ROWS_PER_TEXT_ROW · printCentered · readLineFromKeyboard · scrnBuffer · setColorInversionMask · setInvertedColorsParameters · setScreenBuffer · setScreenBuffer2 · setTextAreaScrollCallback · TEXT_COLS_PER_ROW · TEXT_ROWS_PER_SCREEN · TextAreaScrollCallback · textPosX · textPosY · waitForKeyWithAnimatedCursor · WORDS_PER_SCREEN_ROW

assertf

```
assertf(cond,...)
```

Report an assertion using formatted text. Its visibility and action depend on the build's assertion settings.

pmode4game.h · `-lcardgame`

BYTES_PER_SCREEN

```
BYTES_PER_SCREEN = 0x1800
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

BYTES_PER_SCREEN_ROW

```
BYTES_PER_SCREEN_ROW = 32
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

BYTES_PER_TEXT_ROW

```
BYTES_PER_TEXT_ROW = 256
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

closePMode4Game

```
void closePMode4Game(void);
```

Reserve the aligned 6,144-byte PMODE 4 buffer before opening a graphics game. The open/close pairs own their screen and console state while active. Use screen(0,0) when returning to ordinary 32-column text.

pmode4game.h · `-lcardgame`

concat

```
void concat(char *out, unsigned max, ...);
```

Join the supplied pieces into a bounded result. max includes space for the final zero. Follow the string argument list with a null pointer.

pmode4game.h · `-lcardgame`

drawChar

```
void drawChar(unsigned char c);
```

Draw a character onto the game graphics buffer. The explicit-address form requires writable bytes for each raster row. The stationary forms preserve the text position.

pmode4game.h · `-lcardgame`

drawCharAtAddressWithoutMovingCursor

```
void drawCharAtAddressWithoutMovingCursor(unsigned char c, byte *dest, BOOL drawBottomRow);
```

Draw a character onto the game graphics buffer. The explicit-address form requires writable bytes for each raster row. The stationary forms preserve the text position.

pmode4game.h · `-lcardgame`

drawCharWithoutMovingCursor

```
void drawCharWithoutMovingCursor(unsigned char c);
```

Draw a character onto the game graphics buffer. The explicit-address form requires writable bytes for each raster row. The stationary forms preserve the text position.

pmode4game.h · `-lcardgame`

emulatingPMode4OnCoCo3

```
extern byte emulatingPMode4OnCoCo3;
```

State maintained by the active game renderer. The text position counts character cells and the emulation flag distinguishes the CoCo 3 compatibility display.

pmode4game.h · `-lcardgame`

enableInvertedColorText

```
void enableInvertedColorText(BOOL enabled);
```

Change how later character drawing combines glyph pixels with the background. Transparent spaces preserve pixels already in the game image.

pmode4game.h · `-lcardgame`

getCurrentCharPos

```
byte *getCurrentCharPos(void);
```

Borrow the address of the top-left byte for the current 8×8 glyph in the graphics text buffer.

pmode4game.h · `-lcardgame`

getCursor

```
void getCursor(byte *x, byte *y);
```

Work in character columns and rows, starting at zero. Set or retrieve the coordinates using the active text renderer. HiresTxt and CardGame each export moveCursor; use the intended renderer and link library.

pmode4game.h · `-lcardgame`

GetTimerFunction

```
typedef word (*GetTimerFunction)(void *userdata);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

pmode4game.h · `-lcardgame`

hiResTextConsoleOutHook

```
void hiResTextConsoleOutHook();
```

The game renderer can intercept console output to draw characters in its PMODE 4 buffer. Preserve the previous hook for cleanup.

pmode4game.h · `-lcardgame`

invertCharAtCursor

```
void invertCharAtCursor(void);
```

Toggle the pixels occupied by the cursor cell in the game buffer. The shorter form excluding the bottom raster row preserves that row's graphics.

pmode4game.h · `-lcardgame`

invertCharAtCursorExceptRowBelowGlyph

```
void invertCharAtCursorExceptRowBelowGlyph(void);
```

Toggle the pixels occupied by the cursor cell in the game buffer. The shorter form excluding the bottom raster row preserves that row's graphics.

pmode4game.h · `-lcardgame`

KeyboardPollFunction

```
typedef byte (*KeyboardPollFunction)(void *userdata);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

pmode4game.h · `-lcardgame`

makeSpacesTransparent

```
void makeSpacesTransparent(BOOL transparent);
```

Change how later character drawing combines glyph pixels with the background. Transparent spaces preserve pixels already in the game image.

pmode4game.h · `-lcardgame`

moveCursor

```
void moveCursor(byte x, byte y);
```

Work in character columns and rows, starting at zero. Set or retrieve the coordinates using the active text renderer. HiresTxt and CardGame each export moveCursor; use the intended renderer and link library.

pmode4game.h · `-lcardgame`

moveCursorLeft

```
void moveCursorLeft(byte numColumns);
```

Work in character columns and rows, starting at zero. Set or retrieve the coordinates using the active text renderer. HiresTxt and CardGame each export moveCursor; use the intended renderer and link library.

pmode4game.h · `-lcardgame`

oldCHROUT

```
extern ConsoleOutHook oldCHROUT;
```

The game renderer can intercept console output to draw characters in its PMODE 4 buffer. Preserve the previous hook for cleanup.

pmode4game.h · `-lcardgame`

openPMode4Game

```
void openPMode4Game(void *_scrnBuffer, byte _emulatePMode4OnCoCo3);
```

Reserve the aligned 6,144-byte PMODE 4 buffer before opening a graphics game. The open/close pairs own their screen and console state while active. Use screen(0,0) when returning to ordinary 32-column text.

pmode4game.h · `-lcardgame`

optDelay

```
optDelay(ticks)
```

Target-specific timing convenience for card-game code. Inspect its definition for the selected target before relying on a delay.

pmode4game.h · `-lcardgame`

PIXEL_ROWS_PER_GLYPH

```
PIXEL_ROWS_PER_GLYPH = 7
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

PIXEL_ROWS_PER_SCREEN

```
PIXEL_ROWS_PER_SCREEN = 192
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

PIXEL_ROWS_PER_TEXT_ROW

```
PIXEL_ROWS_PER_TEXT_ROW = 8
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

printCentered

```
byte printCentered(byte row, const char *str);
```

Render a string centered on the requested text row. The return value is the starting column used.

pmode4game.h · `-lcardgame`

readLineFromKeyboard

```
BOOL readLineFromKeyboard(KeyboardPollFunction poll, void *userdata, word ticksPerCursorPeriod,
GetTimerFunction getTimerValue, char *lineBuffer, size_t bufferSize);
```

Supply a polling function, timer callback and their user data. Keep the display buffer alive. For a line, check whether Enter accepted it or Break canceled it.

pmode4game.h · `-lcardgame`

scrnBuffer

```
extern byte *scrnBuffer;
```

Select a different buffer for graphics output. Under CoCo 3 PMODE emulation, the address convention depends on which setter is used; use the header for the active game configuration.

pmode4game.h · `-lcardgame`

setColorInversionMask

```
void setColorInversionMask(byte newColorInversionMask);
```

Change how later character drawing combines glyph pixels with the background. Transparent spaces preserve pixels already in the game image.

pmode4game.h · `-lcardgame`

setInvertedColorsParameters

```
void setInvertedColorsParameters(BOOL drawRowOverGlyph, BOOL drawColumnAtRightOfGlyph);
```

Change how later character drawing combines glyph pixels with the background. Transparent spaces preserve pixels already in the game image.
pmode4game.h · `-lcardgame`

setScreenBuffer

```
void setScreenBuffer(void *screen);
```

Select and display the supplied screen. In CoCo 3 PMODE emulation the address is relative to physical $70000. See setScreenBuffer2 when preparing an offscreen drawing buffer.
pmode4game.h · `-lcardgame`

setScreenBuffer2

```
void setScreenBuffer2(void *screen);
```

Select the library's drawing buffer without changing the video mode or making the buffer visible.
pmode4game.h · `-lcardgame`

setTextAreaScrollCallback

```
void setTextAreaScrollCallback(TextAreaScrollCallback callback, void *userdata);
```

Install the scroll callback and its user data. The user data must remain valid when the renderer calls back.
pmode4game.h · `-lcardgame`

TEXT_COLS_PER_ROW

```
TEXT_COLS_PER_ROW = 32
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.
pmode4game.h · `-lcardgame`

TEXT_ROWS_PER_SCREEN

```
TEXT_ROWS_PER_SCREEN = 24
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.
pmode4game.h · `-lcardgame`

TextAreaScrollCallback

```
typedef void (*TextAreaScrollCallback)(void *userdata);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.
pmode4game.h · `-lcardgame`

textPosX

```
extern byte textPosX;
```

State maintained by the active game renderer. The text position counts character cells and the emulation flag distinguishes the CoCo 3 compatibility display.
pmode4game.h · `-lcardgame`

textPosY

```
extern byte textPosY;
```

State maintained by the active game renderer. The text position counts character cells and the emulation flag distinguishes the CoCo 3 compatibility display.
pmode4game.h · `-lcardgame`

waitForKeyWithAnimatedCursor

```
byte waitForKeyWithAnimatedCursor(KeyboardPollFunction poll, void *userdata, word
ticksPerCursorPeriod, GetTimerFunction getTimerValue);
```

Supply a polling function, timer callback and their user data. Keep the display buffer alive. For a line, check whether Enter accepted it or Break canceled it.

pmode4game.h · `-lcardgame`

WORDS_PER_SCREEN_ROW

```
WORDS_PER_SCREEN_ROW = 16
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

pmode4game.h · `-lcardgame`

8-bit-wide-font.h

Find the game renderer's eight-pixel-wide glyph data. Add **cardgame** under Project Settings → Libraries, or link with `-lcardgame` in your Makefile.

Save the cardgame example for this book

eightBitWideFont

eightBitWideFont

```
extern const unsigned char eightBitWideFont[65][7];
```

Library-owned glyph data. Use the renderer for the font's dimensions and treat the original bytes as read-only.

8-bit-wide-font.h · `-lcardgame`

compiled-cards.h

Find the precomputed suit symbols, ranks and edges used for drawing cards. Add **cardgame** under Project Settings → Libraries, or link with `-lcardgame` in your Makefile.

Save the cardgame example for this book

blackCard_1 · blackCard_10 · blackCard_11 · blackCard_12 · blackCard_13 · blackCard_2 · blackCard_3 · blackCard_4 · blackCard_5 · blackCard_6 · blackCard_7 · blackCard_8 · blackCard_9 · blackCards · blackJokerSuit · blackJokerValue · cardBottomLeft · cardBottomRight · cardLeft · cardRight · cardTopLeft · cardTopRight · redCard_1 · redCard_10 · redCard_11 · redCard_12 · redCard_13 · redCard_2 · redCard_3 · redCard_4 · redCard_5 · redCard_6 · redCard_7 · redCard_8 · redCard_9 · redCards · redJokerSuit · redJokerValue · suit_clubs · suit_diamonds · suit_hearts · suit_spades · suits

blackCard_1

```
extern const unsigned short blackCard_1[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_10

```
extern const unsigned short blackCard_10[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_11

```
extern const unsigned short blackCard_11[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_12

```
extern const unsigned short blackCard_12[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_13

```
extern const unsigned short blackCard_13[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_2

```
extern const unsigned short blackCard_2[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_3

```
extern const unsigned short blackCard_3[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_4

```
extern const unsigned short blackCard_4[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_5

```
extern const unsigned short blackCard_5[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_6

```
extern const unsigned short blackCard_6[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_7

```
extern const unsigned short blackCard_7[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_8

```
extern const unsigned short blackCard_8[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCard_9

```
extern const unsigned short blackCard_9[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackCards

```
extern const unsigned short * const blackCards[13];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackJokerSuit

```
extern const unsigned short blackJokerSuit[7];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

blackJokerValue

```
extern const unsigned short blackJokerValue[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

cardBottomLeft

```
extern const unsigned short cardBottomLeft[5];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

cardBottomRight

```
extern const unsigned short cardBottomRight[5];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

cardLeft

```
extern const unsigned short cardLeft[1];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

cardRight

```
extern const unsigned short cardRight[1];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

cardTopLeft

```
extern const unsigned short cardTopLeft[5];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

cardTopRight

```
extern const unsigned short cardTopRight[5];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_1

```
extern const unsigned short redCard_1[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_10

```
extern const unsigned short redCard_10[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_11

```
extern const unsigned short redCard_11[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_12

```
extern const unsigned short redCard_12[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_13

```
extern const unsigned short redCard_13[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_2

```
extern const unsigned short redCard_2[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_3

```
extern const unsigned short redCard_3[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_4

```
extern const unsigned short redCard_4[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_5

```
extern const unsigned short redCard_5[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_6

```
extern const unsigned short redCard_6[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_7

```
extern const unsigned short redCard_7[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_8

```
extern const unsigned short redCard_8[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCard_9

```
extern const unsigned short redCard_9[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redCards

```
extern const unsigned short * const redCards[13];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redJokerSuit

```
extern const unsigned short redJokerSuit[7];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

redJokerValue

```
extern const unsigned short redJokerValue[10];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

suit_clubs

```
extern const unsigned short suit_clubs[7];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

suit_diamonds

```
extern const unsigned short suit_diamonds[7];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

suit_hearts

```
extern const unsigned short suit_hearts[7];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

suit_spades

```
extern const unsigned short suit_spades[7];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

suits

```
extern const unsigned short * const suits[4];
```

Precomputed graphics used by the card renderer. Treat this library-owned data as read-only.

compiled-cards.h · `-lcardgame`

Ebitor.h

Maintain an editable text buffer and a cursor through terminal callbacks. Add **ebitor** under Project Settings → Libraries, or link with `-lebitor` in your Makefile.

Save the ebitor example for this book

Ebitor · ebitor_clear · ebitor_clrtoeolFunc · Ebitor_findEOL · Ebitor_findSOL · Ebitor_findStringAndScrollToShow · Ebitor_getContentLength · ebitor_getCursorPosFunc · Ebitor_getInstance · Ebitor_getLogicalCursorOffset · Ebitor_getLogicalOffsetAtHome · Ebitor_init · ebitor_KEY_ARROW_DOWN · ebitor_KEY_ARROW_LEFT · ebitor_KEY_ARROW_RIGHT · ebitor_KEY_ARROW_UP · ebitor_KEY_ESCAPE · Ebitor_locateCursorAtSavedPosition · ebitor_locateFunc · ebitor_printFunc · ebitor_printPrefixFunc · Ebitor_processEditing · Ebitor_processKey · Ebitor_processUpOrDownArrowsAndRefresh · Ebitor_saveCursorPosition · ebitor_scrollTerminalDownFunc · ebitor_scrollTerminalUpFunc · Ebitor_scrollToShowLogicalOffset · Ebitor_setContentLength · Ebitor_setLogicalCursorAtBeginning · Ebitor_shutdown · ebitor_waitKeyFunc · INSERT_BUFFER_SIZE · InsertBuffer

Ebitor

```
typedef struct Ebitor { … } Ebitor;
```

Supply a persistent writable text buffer, its byte capacity, the terminal dimensions, and compatible screen and key callbacks. The editor manages one instance. Release it with shutdown before repurposing its buffer.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_clear

```
typedef void (*ebitor_clear)(void);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

ebitor_clrtoeolFunc

```
typedef void (*ebitor_clrtoeolFunc)(void);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

Ebitor_findEOL

```
size_t Ebitor_findEOL(size_t startOffset);
```

Locate the byte offset of the line beginning or its delimiter from a position within the text. The last unterminated line ends at the filled buffer length.

Ebitor.h · `-lebitor`

Ebitor_findSOL

```
size_t Ebitor_findSOL(size_t startOffset);
```

Locate the byte offset of the line beginning or its delimiter from a position within the text. The last unterminated line ends at the filled buffer length.

Ebitor.h · `-lebitor`

Ebitor_findStringAndScrollToShow

```
BOOL Ebitor_findStringAndScrollToShow(size_t startLogicalOffset, const char *strSearchedFor, size_t *foundLogicalOffset);
```

Search for a nonempty zero-terminated string starting at a byte offset, or display a known range. The search writes the found offset and scrolls only upon success.

Ebitor.h · `-lebitor`

Ebitor_getContentLength

```
size_t Ebitor_getContentLength(void);
```

After filling the text buffer, tell the editor how many bytes contain text. The length must fit the capacity supplied at init. Setting it repositions the cursor at the beginning.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_getCursorPosFunc

```
typedef void (*ebitor_getCursorPosFunc)(byte *column, byte *row);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

Ebitor_getInstance

```
Ebitor *Ebitor_getInstance(void);
```

Borrow the singleton editor state for debugging its internals. Use the public editing functions for normal interaction.

Ebitor.h · `-lebitor`

Ebitor_getLogicalCursorOffset

```
size_t Ebitor_getLogicalCursorOffset(void);
```

Distinguish byte offsets within the entire document from the on-screen row and column. Home in the current window can correspond to text partway through a file.

Ebitor.h · `-lebitor`

Ebitor_getLogicalOffsetAtHome

```
size_t Ebitor_getLogicalOffsetAtHome(byte cursorCol, byte cursorRow);
```

Distinguish byte offsets within the entire document from the on-screen row and column. Home in the current window can correspond to text partway through a file.

Ebitor.h · `-lebitor`

Ebitor_init

```
void Ebitor_init(char *bufferBegin, size_t bufferSize, byte termWidth, byte termHeight,
ebitor_clear clear, ebitor_locateFunc locate, ebitor_getCursorPosFunc getCursorPos,
ebitor_printFunc print, ebitor_printPrefixFunc printPrefix, ebitor_clrtoeolFunc clrtoeol,
ebitor_scrollTerminalUpFunc scrollTerminalUp, ebitor_scrollTerminalDownFunc scrollTerminalDown,
ebitor_waitKeyFunc waitKey);
```

Supply a persistent writable text buffer, its byte capacity, the terminal dimensions, and compatible screen and key callbacks. The editor manages one instance. Release it with shutdown before repurposing its buffer.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_KEY_ARROW_DOWN

```
ebitor_KEY_ARROW_DOWN = 259
```

These codes start above the byte range. Keep the terminal key in a word so the editor can distinguish a command from a printable character. Handle escape outside Ebitor_processKey.

Ebitor.h · `-lebitor`

ebitor_KEY_ARROW_LEFT

```
ebitor_KEY_ARROW_LEFT = 256
```

These codes start above the byte range. Keep the terminal key in a word so the editor can distinguish a command from a printable character. Handle escape outside Ebitor_processKey.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_KEY_ARROW_RIGHT

```
ebitor_KEY_ARROW_RIGHT = 257
```

These codes start above the byte range. Keep the terminal key in a word so the editor can distinguish a command from a printable character. Handle escape outside Ebitor_processKey.

Ebitor.h · `-lebitor`

ebitor_KEY_ARROW_UP

```
ebitor_KEY_ARROW_UP = 258
```

These codes start above the byte range. Keep the terminal key in a word so the editor can distinguish a command from a printable character. Handle escape outside Ebitor_processKey.

Ebitor.h · `-lebitor`

ebitor_KEY_ESCAPE

```
ebitor_KEY_ESCAPE = 260
```

These codes start above the byte range. Keep the terminal key in a word so the editor can distinguish a command from a printable character. Handle escape outside Ebitor_processKey.

Ebitor.h · `-lebitor`

Ebitor_locateCursorAtSavedPosition

```
void Ebitor_locateCursorAtSavedPosition(void);
```

Save or restore the current terminal location around a series of programmatically supplied keys.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_locateFunc

```
typedef void (*ebitor_locateFunc)(byte column, byte row);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

ebitor_printFunc

```
typedef BOOL (*ebitor_printFunc)(const char *str);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

ebitor_printPrefixFunc

```
typedef BOOL (*ebitor_printPrefixFunc)(const char *str, size_t prefixLength);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

Ebitor_processEditing

```
void Ebitor_processEditing(BOOL startWithRefresh);
```

Run the editing loop through the installed input and output callbacks. startWithRefresh requests a repaint. Returning from the loop saves the current position.

Ebitor.h · `-lebitor`

Ebitor_processKey

```
void Ebitor_processKey(word key);
```

Apply one printable character or supported navigation code. Restore the saved cursor before a sequence and save it again afterward. Handle the escape request outside this routine.

Ebitor.h · `-lebitor` · See it used in editlog.c

Ebitor_processUpOrDownArrowsAndRefresh

```
void Ebitor_processUpOrDownArrowsAndRefresh(BOOL isUpArrow, size_t numRepetitions);
```

Move up or down by the specified number of repeats and refresh once. Use this for a large move to avoid repainting on each simulated key.

Ebitor.h · `-lebitor`

Ebitor_saveCursorPosition

```
void Ebitor_saveCursorPosition(void);
```

Save or restore the current terminal location around a series of programmatically supplied keys.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_scrollTerminalDownFunc

```
typedef void (*ebitor_scrollTerminalDownFunc)(byte numRows);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

ebitor_scrollTerminalUpFunc

```
typedef void (*ebitor_scrollTerminalUpFunc)(byte numRows);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

Ebitor.h · `-lebitor`

Ebitor_scrollToShowLogicalOffset

```
void Ebitor_scrollToShowLogicalOffset(size_t logicalOffsetToShow, size_t lengthToShow);
```

Search for a nonempty zero-terminated string starting at a byte offset, or display a known range. The search writes the found offset and scrolls only upon success.

Ebitor.h · `-lebitor`

Ebitor_setContentLength

```
void Ebitor_setContentLength(size_t newLength);
```

After filling the text buffer, tell the editor how many bytes contain text. The length must fit the capacity supplied at init. Setting it repositions the cursor at the beginning.

Ebitor.h · `-lebitor` · See it used in editlog.c

Ebitor_setLogicalCursorAtBeginning

```
void Ebitor_setLogicalCursorAtBeginning(void);
```

Distinguish byte offsets within the entire document from the on-screen row and column. Home in the current window can correspond to text partway through a file.

Ebitor.h · `-lebitor`

Ebitor_shutdown

```
void Ebitor_shutdown(void);
```

Supply a persistent writable text buffer, its byte capacity, the terminal dimensions, and compatible screen and key callbacks. The editor manages one instance. Release it with shutdown before repurposing its buffer.

Ebitor.h · `-lebitor` · See it used in editlog.c

ebitor_waitKeyFunc

```
typedef word (*ebitor_waitKeyFunc)(void);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.
Ebitor.h · `-lebitor`

INSERT_BUFFER_SIZE

```
INSERT_BUFFER_SIZE = 64
```

Ebitor's internal insert buffer stages pending characters. For application text length use the content-length functions.
Ebitor.h · `-lebitor`

InsertBuffer

```
typedef struct InsertBuffer { … } InsertBuffer;
```

Ebitor's internal insert buffer stages pending characters. For application text length use the content-length functions.
Ebitor.h · `-lebitor`

Window.h

Describe the guest text window used by the Ebitor terminal adapters. Add **ebitor** under Project Settings → Libraries, or link with `-lebitor` in your Makefile.
Save the ebitor example for this book
ebitor_Window

ebitor_Window

```
typedef struct ebitor_Window { … } ebitor_Window;
```

Describe the guest-side text window served by the terminal adapter. The GIME window also supplies normal and inverted character attributes.
Window.h · `-lebitor` · See it used in editlog.c

sg4term.h

Implement terminal operations on a traditional CoCo character screen. Add **ebitor** under Project Settings → Libraries, or link with `-lebitor` in your Makefile.
Save the ebitor example for this book
sg4term_clear · sg4term_clrtoeol · sg4term_convertASCIIToScreenByte · sg4term_eraseCurrentLine · sg4term_getChars · sg4term_getCursorPos · sg4term_init · sg4term_invertColors · sg4term_locate · sg4term_print · sg4term_printPrefix · sg4term_scrollWindowDown · sg4term_scrollWindowUp · sg4term_waitEditorKey

sg4term_clear

```
void sg4term_clear(void);
```

Fill the current terminal window with spaces. Erasing the view and moving the cursor are separate operations.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_clrtoeol

```
void sg4term_clrtoeol(void);
```

Clear from the current position to the end, or erase the whole line and position the cursor at its start.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_convertASCIIToScreenByte

```
char sg4term_convertASCIIToScreenByte(char c);
```

Translate an ASCII character to the compatible screen encoding, or copy a specified number of cells from a screen position. Size the destination for len bytes.
sg4term.h · `-lebitor`

sg4term_eraseCurrentLine

```
void sg4term_eraseCurrentLine(void);
```

Clear from the current position to the end, or erase the whole line and position the cursor at its start.
sg4term.h · `-lebitor`

sg4term_getChars

```
void sg4term_getChars(byte column, byte row, char *dest, size_t len);
```

Translate an ASCII character to the compatible screen encoding, or copy a specified number of cells from a screen position. Size the destination for len bytes.
sg4term.h · `-lebitor`

sg4term_getCursorPos

```
void sg4term_getCursorPos(byte *column, byte *row);
```

Supply or read a zero-based column and row relative to the terminal window. Check bounds in the caller.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_init

```
void sg4term_init(char *screenBuffer, const ebitor_Window *window, byte (*inkeyFuncPtr)(void
*cookie), void *inkeyFuncCookie, word *timerPtr);
```

Prepare an existing screen with a window, a nonblocking key callback, caller data and a timer ticking at 60 Hz. GIME works with hardware text cells; SG4 uses the character screen. Choose and allocate the proper display before initialization.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_invertColors

```
void sg4term_invertColors(void);
```

Choose the character colors for later writes. In GIME mode each character has a separate attribute byte. Attribute setters return the previous value.
sg4term.h · `-lebitor`

sg4term_locate

```
void sg4term_locate(byte column, byte row);
```

Supply or read a zero-based column and row relative to the terminal window. Check bounds in the caller.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_print

```
BOOL sg4term_print(const char *str);
```

Write inside the configured terminal window. A false result means output reached the bottom. Scroll and reposition explicitly. For a prefix supply the number of valid input characters.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_printPrefix

```
BOOL sg4term_printPrefix(const char *str, size_t prefixLength);
```

Write inside the configured terminal window. A false result means output reached the bottom. Scroll and reposition explicitly. For a prefix supply the number of valid input characters.
sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_scrollWindowDown

```
void sg4term_scrollWindowDown(byte numRows);
```

Shift the terminal window's contents by a number of **text rows**. The caller chooses the direction and then positions the cursor as appropriate.

sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_scrollWindowUp

```
void sg4term_scrollWindowUp(byte numRows);
```

Shift the terminal window's contents by a number of **text rows**. The caller chooses the direction and then positions the cursor as appropriate.

sg4term.h · `-lebitor` · See it used in editlog.c

sg4term_waitEditorKey

```
word sg4term_waitEditorKey(void);
```

Block for keyboard input. Editor keys may exceed byte range, so keep the word result for arrow and editing codes. The simple GIME waitKey returns a byte.

sg4term.h · `-lebitor` · See it used in editlog.c

gimeterm.h

Implement terminal operations using the CoCo 3 GIME hardware text screen. Add **ebitor** under Project Settings → Libraries, or link with `-lebitor` in your Makefile.

Save the ebitor example for this book

gimeterm_clear · gimeterm_clrtoeol · gimeterm_eraseCurrentLine · gimeterm_getCursorPos · gimeterm_getWindow · gimeterm_init · gimeterm_invertColors · gimeterm_locate · gimeterm_newline · gimeterm_print · gimeterm_printPrefix · gimeterm_putchar · gimeterm_scrollWindowDown · gimeterm_scrollWindowUp · gimeterm_setInvertedTextAttribute · gimeterm_setTextAttribute · gimeterm_setWindow · gimeterm_waitEditorKey · gimeterm_waitKey · gimeterm_Window

gimeterm_clear

```
void gimeterm_clear(void);
```

Fill the current terminal window with spaces. Erasing the view and moving the cursor are separate operations.

gimeterm.h · `-lebitor`

gimeterm_clrtoeol

```
void gimeterm_clrtoeol(void);
```

Clear from the current position to the end, or erase the whole line and position the cursor at its start.

gimeterm.h · `-lebitor`

gimeterm_eraseCurrentLine

```
void gimeterm_eraseCurrentLine(void);
```

Clear from the current position to the end, or erase the whole line and position the cursor at its start.

gimeterm.h · `-lebitor`

gimeterm_getCursorPos

```
void gimeterm_getCursorPos(byte *column, byte *row);
```

Supply or read a zero-based column and row relative to the terminal window. Check bounds in the caller.

gimeterm.h · `-lebitor`

gimeterm_getWindow

```
void gimeterm_getWindow(gimeterm_Window *dest);
```

Copy the active GIME window state to caller storage or install saved state. Coordinates in later terminal operations are relative to its origin.
gimeterm.h · `-lebitor`

gimeterm_init

```
void gimeterm_init(void *screenBuffer, byte screenMode, const gimeterm_Window *window, byte (*inkeyFuncPtr)(void *cookie), void *inkeyFuncCookie, word *timerPtr);
```

Prepare an existing screen with a window, a nonblocking key callback, caller data and a timer ticking at 60 Hz. GIME works with hardware text cells; SG4 uses the character screen. Choose and allocate the proper display before initialization.
gimeterm.h · `-lebitor`

gimeterm_invertColors

```
void gimeterm_invertColors(void);
```

Choose the character colors for later writes. In GIME mode each character has a separate attribute byte. Attribute setters return the previous value.
gimeterm.h · `-lebitor`

gimeterm_locate

```
void gimeterm_locate(byte column, byte row);
```

Supply or read a zero-based column and row relative to the terminal window. Check bounds in the caller.
gimeterm.h · `-lebitor`

gimeterm_newline

```
BOOL gimeterm_newline(void);
```

Write inside the configured terminal window. A false result means output reached the bottom. Scroll and reposition explicitly. For a prefix supply the number of valid input characters.
gimeterm.h · `-lebitor`

gimeterm_print

```
BOOL gimeterm_print(const char *str);
```

Write inside the configured terminal window. A false result means output reached the bottom. Scroll and reposition explicitly. For a prefix supply the number of valid input characters.
gimeterm.h · `-lebitor`

gimeterm_printPrefix

```
BOOL gimeterm_printPrefix(const char *str, size_t prefixLength);
```

Write inside the configured terminal window. A false result means output reached the bottom. Scroll and reposition explicitly. For a prefix supply the number of valid input characters.
gimeterm.h · `-lebitor`

gimeterm_putchar

```
BOOL gimeterm_putchar(int c);
```

Write inside the configured terminal window. A false result means output reached the bottom. Scroll and reposition explicitly. For a prefix supply the number of valid input characters.
gimeterm.h · `-lebitor`

gimeterm_scrollWindowDown

```
void gimeterm_scrollWindowDown(byte numRows);
```

Shift the terminal window's contents by a number of **text rows**. The caller chooses the direction and then positions the cursor as appropriate.

gimeterm.h · `-lebitor`

gimeterm_scrollWindowUp

```
void gimeterm_scrollWindowUp(byte numRows);
```

Shift the terminal window's contents by a number of **text rows**. The caller chooses the direction and then positions the cursor as appropriate.

gimeterm.h · `-lebitor`

gimeterm_setInvertedTextAttribute

```
byte gimeterm_setInvertedTextAttribute(byte newAttribute);
```

Choose the character colors for later writes. In GIME mode each character has a separate attribute byte. Attribute setters return the previous value.

gimeterm.h · `-lebitor`

gimeterm_setTextAttribute

```
byte gimeterm_setTextAttribute(byte newAttribute);
```

Choose the character colors for later writes. In GIME mode each character has a separate attribute byte. Attribute setters return the previous value.

gimeterm.h · `-lebitor`

gimeterm_setWindow

```
void gimeterm_setWindow(const gimeterm_Window *state);
```

Copy the active GIME window state to caller storage or install saved state. Coordinates in later terminal operations are relative to its origin.

gimeterm.h · `-lebitor`

gimeterm_waitEditorKey

```
word gimeterm_waitEditorKey(void);
```

Block for keyboard input. Editor keys may exceed byte range, so keep the word result for arrow and editing codes. The simple GIME waitKey returns a byte.

gimeterm.h · `-lebitor`

gimeterm_waitKey

```
byte gimeterm_waitKey(void);
```

Block for keyboard input. Editor keys may exceed byte range, so keep the word result for arrow and editing codes. The simple GIME waitKey returns a byte.

gimeterm.h · `-lebitor`

gimeterm_Window

```
typedef struct gimeterm_Window { … } gimeterm_Window;
```

Describe the guest-side text window served by the terminal adapter. The GIME window also supplies normal and inverted character attributes.

gimeterm.h · `-lebitor`

hirestxt.h

Render text and a cursor into a graphics screen. Add **hirestxt** under Project Settings → Libraries, or link with `-lhirestxt` in your Makefile.

Save the hirestxt example for this book

animateCursor · BYTES_PER_PIXEL_ROW · clear · clearRowsToEOS · closeHiResTextScreen · clrscr · clrtobot · clrtoeol · getCursorColumn · getCursorRow · HIRESHEIGHT · HiResTextScreenInit · HiResTextScreenInit2 · home ·

initHiResTextScreen · initHiResTextScreen2 · initVT52 · invertPixelsAtCursor · moveCursor · PIXEL_COLS_PER_SCREEN · PIXEL_ROWS_PER_SCREEN · PIXEL_ROWS_PER_TEXT_ROW · processConsoleOutChar · removeCursor · scrollTextScreenUp · setBackgroundColor · setBoldMode · setForegroundBoldColor · setForegroundColor · setInverseVideoMode · waitKeyBlinkingCursor · writeCenteredLine · writeChar · writeCharAt · writeCharAt_320x16 · writeCharAt_42cols · writeCharAt_51cols · WriteCharAtFuncPtr · writeDecWord · writeString

animateCursor

```
void animateCursor(void);
```

Update the software cursor from the configured timer, remove it after animation, or wait for a key while it blinks. The default input function expects BASIC.

hirestxt.h · `-lhirestxt`

BYTES_PER_PIXEL_ROW

```
BYTES_PER_PIXEL_ROW = 4
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

hirestxt.h · `-lhirestxt`

clear

```
void clear(void);
```

Erase part or all of the graphics text buffer. For explicit rows use a raster fill byte and a zero-based text row. Choose the narrower operation if surrounding graphics should survive.

hirestxt.h · `-lhirestxt`

clearRowsToEOS

```
void clearRowsToEOS(byte byteToClearWith, byte textRow);
```

Erase part or all of the graphics text buffer. For explicit rows use a raster fill byte and a zero-based text row. Choose the narrower operation if surrounding graphics should survive.

hirestxt.h · `-lhirestxt`

closeHiResTextScreen

```
void closeHiResTextScreen(void);
```

Draw text into a caller-owned graphics buffer. Choose a renderer for the actual width, depth and font. For 320×192×16 and 64 text columns use the extended initializer. If printf is redirected, finish with closeHiResTextScreen to restore the hook.

hirestxt.h · `-lhirestxt` · See it used in glyph.c

clrscr

```
void clrscr(void);
```

Erase part or all of the graphics text buffer. For explicit rows use a raster fill byte and a zero-based text row. Choose the narrower operation if surrounding graphics should survive.

hirestxt.h · `-lhirestxt`

clrtobot

```
void clrtobot(void);
```

Erase part or all of the graphics text buffer. For explicit rows use a raster fill byte and a zero-based text row. Choose the narrower operation if surrounding graphics should survive.

hirestxt.h · `-lhirestxt`

clrtoeol

```
void clrtoeol(void);
```

Erase part or all of the graphics text buffer. For explicit rows use a raster fill byte and a zero-based text row. Choose the narrower operation if surrounding graphics should survive.

hirestxt.h · `-lhirestxt`

getCursorColumn

```
byte getCursorColumn(void);
```

Read the current character position or move it to the origin. Screen coordinates here count character cells, independent of the underlying pixel width.

hirestxt.h · `-lhirestxt`

getCursorRow

```
byte getCursorRow(void);
```

Read the current character position or move it to the origin. Screen coordinates here count character cells, independent of the underlying pixel width.

hirestxt.h · `-lhirestxt`

HIRESHEIGHT

```
HIRESHEIGHT = 0
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

hirestxt.h · `-lhirestxt`

HiResTextScreenInit

```
struct HiResTextScreenInit
```

Draw text into a caller-owned graphics buffer. Choose a renderer for the actual width, depth and font. For 320×192×16 and 64 text columns use the extended initializer. If printf is redirected, finish with closeHiResTextScreen to restore the hook.

hirestxt.h · `-lhirestxt`

HiResTextScreenInit2

```
struct HiResTextScreenInit2
```

Draw text into a caller-owned graphics buffer. Choose a renderer for the actual width, depth and font. For 320×192×16 and 64 text columns use the extended initializer. If printf is redirected, finish with closeHiResTextScreen to restore the hook.

hirestxt.h · `-lhirestxt` · See it used in glyph.c

home

```
home()
```

Read the current character position or move it to the origin. Screen coordinates here count character cells, independent of the underlying pixel width.

hirestxt.h · `-lhirestxt`

initHiResTextScreen

```
void initHiResTextScreen(const struct HiResTextScreenInit *init);
```

Draw text into a caller-owned graphics buffer. Choose a renderer for the actual width, depth and font. For 320×192×16 and 64 text columns use the extended initializer. If printf is redirected, finish with closeHiResTextScreen to restore the hook.

hirestxt.h · `-lhirestxt`

initHiResTextScreen2

```
void initHiResTextScreen2(const struct HiResTextScreenInit2 *init);
```

Draw text into a caller-owned graphics buffer. Choose a renderer for the actual width, depth and font. For 320×192×16 and 64 text columns use the extended initializer. If printf is redirected, finish with closeHiResTextScreen to restore the hook.

hirestxt.h · `-lhirestxt` · See it used in glyph.c

initVT52

```
void initVT52(void);
```

Initialize the escape-sequence parser and feed it one console character at a time. Control sequences can change the cursor, clear cells and change how later text is rendered.

hirestxt.h · `-lhirestxt`

invertPixelsAtCursor

```
void invertPixelsAtCursor(void);
```

Invert the pixels at the cursor, or scroll the graphics text by one text row. Both modify the buffer in place.

hirestxt.h · `-lhirestxt`

moveCursor

```
void moveCursor(byte x, byte y);
```

Work in character columns and rows, starting at zero. Set or retrieve the coordinates using the active text renderer. HiresTxt and CardGame each export moveCursor; use the intended renderer and link library.

hirestxt.h · `-lhirestxt`

PIXEL_COLS_PER_SCREEN

```
PIXEL_COLS_PER_SCREEN = 1
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

hirestxt.h · `-lhirestxt`

PIXEL_ROWS_PER_SCREEN

```
PIXEL_ROWS_PER_SCREEN = 2
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

hirestxt.h · `-lhirestxt`

PIXEL_ROWS_PER_TEXT_ROW

```
PIXEL_ROWS_PER_TEXT_ROW = 3
```

Size or stride for the renderer named below. A pixel row, a text row and a byte count measure different parts of the screen. Use the constant from the same header as your drawing function.

hirestxt.h · `-lhirestxt`

processConsoleOutChar

```
void processConsoleOutChar(byte ch);
```

Initialize the escape-sequence parser and feed it one console character at a time. Control sequences can change the cursor, clear cells and change how later text is rendered.

hirestxt.h · `-lhirestxt`

removeCursor

```
void removeCursor(void);
```

Update the software cursor from the configured timer, remove it after animation, or wait for a key while it blinks. The default input function expects BASIC.
hirestxt.h · `-lhirestxt`

scrollTextScreenUp

```
void scrollTextScreenUp(void);
```

Invert the pixels at the cursor, or scroll the graphics text by one text row. Both modify the buffer in place.
hirestxt.h · `-lhirestxt`

setBackgroundColor

```
void setBackgroundColor(byte color);
```

Choose the style for future rendered characters. In the four-bit graphics mode, normal foreground, bold foreground and background are palette indices 0–15.
hirestxt.h · `-lhirestxt`

setBoldMode

```
void setBoldMode(BOOL bold);
```

Choose the style for future rendered characters. In the four-bit graphics mode, normal foreground, bold foreground and background are palette indices 0–15.
hirestxt.h · `-lhirestxt`

setForegroundBoldColor

```
void setForegroundBoldColor(byte color);
```

Choose the style for future rendered characters. In the four-bit graphics mode, normal foreground, bold foreground and background are palette indices 0–15.
hirestxt.h · `-lhirestxt`

setForegroundColor

```
void setForegroundColor(byte color);
```

Choose the style for future rendered characters. In the four-bit graphics mode, normal foreground, bold foreground and background are palette indices 0–15.
hirestxt.h · `-lhirestxt`

setInverseVideoMode

```
void setInverseVideoMode(BOOL invert);
```

Choose the style for future rendered characters. In the four-bit graphics mode, normal foreground, bold foreground and background are palette indices 0–15.
hirestxt.h · `-lhirestxt`

waitKeyBlinkingCursor

```
byte waitKeyBlinkingCursor(void);
```

Update the software cursor from the configured timer, remove it after animation, or wait for a key while it blinks. The default input function expects BASIC.
hirestxt.h · `-lhirestxt`

writeCenteredLine

```
void writeCenteredLine(byte row, const char *line);
```

Draw text in the current software-rendered screen. writeString stops at the first zero, and writeCenteredLine centers on a zero-based text row. writeDecWord produces decimal digits.
hirestxt.h · `-lhirestxt`

writeChar

```
void writeChar(byte ch);
```

Draw text in the current software-rendered screen. writeString stops at the first zero, and writeCenteredLine centers on a zero-based text row. writeDecWord produces decimal digits.

hirestxt.h · `-lhirestxt`

writeCharAt

```
void writeCharAt(byte x, byte y, byte asciiCode);
```

Write a glyph at a zero-based character cell in the initialized graphics buffer. Select the width-specific raster routine when preparing the renderer. The 320×16 routine is for 64 columns of software text.

hirestxt.h · `-lhirestxt`

writeCharAt_320x16

```
void writeCharAt_320x16(byte x, byte y, byte asciiCode);
```

Write a glyph at a zero-based character cell in the initialized graphics buffer. Select the width-specific raster routine when preparing the renderer. The 320×16 routine is for 64 columns of software text.

hirestxt.h · `-lhirestxt`

writeCharAt_42cols

```
void writeCharAt_42cols(byte x, byte y, byte asciiCode);
```

Write a glyph at a zero-based character cell in the initialized graphics buffer. Select the width-specific raster routine when preparing the renderer. The 320×16 routine is for 64 columns of software text.

hirestxt.h · `-lhirestxt`

writeCharAt_51cols

```
void writeCharAt_51cols(byte x, byte y, byte asciiCode);
```

Write a glyph at a zero-based character cell in the initialized graphics buffer. Select the width-specific raster routine when preparing the renderer. The 320×16 routine is for 64 columns of software text.

hirestxt.h · `-lhirestxt` · See it used in glyph.c

WriteCharAtFuncPtr

```
typedef void (*WriteCharAtFuncPtr)(byte x, byte y, byte asciiCode);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

hirestxt.h · `-lhirestxt`

writeDecWord

```
void writeDecWord(word w);
```

Draw text in the current software-rendered screen. writeString stops at the first zero, and writeCenteredLine centers on a zero-based text row. writeDecWord produces decimal digits.

hirestxt.h · `-lhirestxt`

writeString

```
void writeString(const char *str);
```

Draw text in the current software-rendered screen. writeString stops at the first zero, and writeCenteredLine centers on a zero-based text row. writeDecWord produces decimal digits.

hirestxt.h · `-lhirestxt` · See it used in glyph.c

font4x8.h

Find the narrow four-pixel glyph data for HiresTxt. Add **hirestxt** under Project Settings → Libraries, or link with `-lhirestxt` in your Makefile.

Save the hirestxt example for this book

font4x8

font4x8

```
extern const unsigned char font4x8[1536];
```

Library-owned glyph data. Use the renderer for the font's dimensions and treat the original bytes as read-only.

font4x8.h · `-lhirestxt`

font5x8.h

Find the five-pixel glyph data for HiresTxt. Add **hirestxt** under Project Settings → Libraries, or link with `-lhirestxt` in your Makefile.

Save the hirestxt example for this book

font5x8

font5x8

```
extern const unsigned char font5x8[1536];
```

Library-owned glyph data. Use the renderer for the font's dimensions and treat the original bytes as read-only.

font5x8.h · `-lhirestxt`

wydealloc.h

Manage variable-size allocations inside a buffer your program owns. Add **wydealloc** under Project Settings → Libraries, or link with `-lwydealloc` in your Makefile.

Save the wydealloc example for this book

AllocatedBlockFunctor · MemPool · MemPool_alloc · MemPool_free · MemPool_getPoolSize · MemPool_getStats · MemPool_init · MemPool_owns

AllocatedBlockFunctor

```
typedef void (*AllocatedBlockFunctor)(const void *clientBlockAddress, unsigned short
clientBlockSizeInBytes, void *userData);
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

wydealloc.h · `-lwydealloc`

MemPool

```
typedef struct MemPool { … } MemPool;
```

Give the pool a persistent byte array, passing its first byte and the address just past the last one. Initialization returns false if the region is too small. The calling program owns this backing memory.

wydealloc.h · `-lwydealloc` · See it used in pool.c

MemPool_alloc

```
void *MemPool_alloc(MemPool *pool, unsigned short numBytesRequested);
```

Allocate from your initialized pool. NULL means insufficient space. A zero-sized request is treated as one byte. Successful bytes are uninitialized. Release with the same pool; freeing NULL does nothing.

wydealloc.h · `-lwydealloc` · See it used in pool.c

MemPool_free

```
void MemPool_free(MemPool *pool, void *ptr);
```

Allocate from your initialized pool. NULL means insufficient space. A zero-sized request is treated as one byte. Successful bytes are uninitialized. Release with the same pool; freeing NULL does nothing.

wydealloc.h · `-lwydealloc` · See it used in pool.c

MemPool_getPoolSize

```
unsigned short MemPool_getPoolSize(const MemPool *pool);
```

Query whether an address lies in the pool, the size of its backing region, or its usage statistics. An optional callback visits blocks that remain allocated.

wydealloc.h · `-lwydealloc`

MemPool_getStats

```
void MemPool_getStats(const MemPool *pool, unsigned short *numAllocatedBlocks, unsigned short
*numFreeBlocks, unsigned short *totalAllocatedBytes, unsigned short *totalFreeBytes, unsigned short
*largestFreeBlock, AllocatedBlockFunctor allocatedBlockFunctor, void
*allocatedBlockFunctorUserData);
```

Query whether an address lies in the pool, the size of its backing region, or its usage statistics. An optional callback visits blocks that remain allocated.

wydealloc.h · `-lwydealloc`

MemPool_init

```
BOOL MemPool_init(MemPool *pool, void *start, void *end);
```

Give the pool a persistent byte array, passing its first byte and the address just past the last one. Initialization returns false if the region is too small. The calling program owns this backing memory.

wydealloc.h · `-lwydealloc` · See it used in pool.c

MemPool_owns

```
BOOL MemPool_owns(MemPool *pool, void *ptr);
```

Query whether an address lies in the pool, the size of its backing region, or its usage statistics. An optional callback visits blocks that remain allocated.

wydealloc.h · `-lwydealloc`

decbfile.h

Read and write named Disk BASIC files, sectors and fixed-size records. Add **decbfile** under Project Settings → Libraries, or link with `-ldecbfile` in your Makefile.

For a worked route from drive registration through a file read and cleanup, open Follow a word through FD0. See also where a file lives inside a floppy image.

Save the decbfile example for this book

decb_closeDir · decb_closeRecordFile · decb_closeSectorFile · decb_computeTrackAndSector · decb_copyFile · decb_createRecordFile · decb_createSectorFile · decb_denormalizeFilename · decb_dskcon · DECB_DSKCON_READ · DECB_DSKCON_WRITE · DECB_ERR_ALREADY_EXISTS · DECB_ERR_CORRUPT_DIR · DECB_ERR_CORRUPT_FAT · DECB_ERR_DRIVE_NOT_REGISTERED · DECB_ERR_END_OF_DIR · DECB_ERR_FAT_NOT_LOADED · DECB_ERR_INTERRUPTED · DECB_ERR_INVALID_ARGUMENT · DECB_ERR_IO · DECB_ERR_NOT_FOUND · DECB_ERR_OUT_OF_SPACE · decb_findDirEntry · decb_findFreeGranule · decb_flush · DECB_FORMAT_ASCII · DECB_FORMAT_BINARY · decb_freeGranuleChain · decb_getFileGranuleFromIndex · decb_getFileSizeFromDirEntry · decb_getFileSizeFromFilename · decb_getFreeSpaceAddress · decb_getFreeSpaceSize · decb_getNumFreeGranules · decb_getNumGranulesInOpenFile · decb_getNumRecords · decb_getOpenFileSize · decb_hasFullSector · decb_init · decb_initSeqWriteBuffer · decb_isValidFATEntry · decb_kill · DECB_LAST_DIR_SECTOR ·

DECB_MAX_NUM_GRANULES · DECB_MAX_NUM_TRACKS · decb_normalizeFilename · DECB_OK · decb_openDir · decb_openRecordFile · decb_openSectorFile · decb_openSectorFileFromDirEntry · decb_readDir · decb_readFAT · decb_readHeader · decb_readRecord · decb_readSector · decb_registerDrive · decb_registerWrittenBytes · decb_rename · DECB_SEQ_WRITE_BUFFER_SIZE · decb_setDskConAddresses · decb_setFATEntry · decb_setNumBytesUsedInLastSector · decb_shutdown · decb_truncateOpenFile · decb_truncateOpenRecordFile · DECB_TYPE_ASCII_TEXT · DECB_TYPE_BASIC_DATA · DECB_TYPE_BASIC_PROGRAM · DECB_TYPE_MACHINE_CODE · decb_unregisterAllDrives · decb_unregisterDrive · decb_writeFAT · decb_writeHeader · decb_writeRecord · decb_writeSector · DECBDirEntry · DECBDirIterator · DECBDrive · DECBDskConVariables · DECBFile · DECBRecordFile · DECBSeqWriteBuffer

decb_closeDir

```
byte decb_closeDir(DECBDirIterator *dirIter);
```

Keep the iterator for the duration of a directory scan. Each successful read yields a pointer to an entry in its buffer. Copy an entry before advancing if you need it later. Stop on DECB_ERR_END_OF_DIR or another error, then close.

decbfile.h · `-ldecbfile`

decb_closeRecordFile

```
byte decb_closeRecordFile(DECBRecordFile *recFile);
```

Truncate at a record boundary, then close the open record file to complete updates. Check each status independently.

decbfile.h · `-ldecbfile`

decb_closeSectorFile

```
byte decb_closeSectorFile(DECBFile *file);
```

Close an open sector file, completing any pending directory metadata update. Check the returned DECB status. A write can succeed before a later close fails.

decbfile.h · `-ldecbfile` · See it used in disk.c

decb_computeTrackAndSector

```
byte decb_computeTrackAndSector(DECBFile *file, byte fileGranIndex, byte sectorIndexInGranule, byte *track, byte *sector);
```

Translate a file-relative granule index using the open file's allocation chain. Within a granule the sector offset is 0–8. Check the DECB status before using the physical coordinates.

decbfile.h · `-ldecbfile`

decb_copyFile

```
byte decb_copyFile(byte srcDriveNo, DECBDirEntry *srcDirEntry, byte destDriveNo, const char *destFilename, byte (*progressFunctor)(word currentSectorIndex, word totalNumSectors, void *userData), void *userData);
```

Copy an entry to a destination name, optionally reporting progress after sectors. A callback that returns zero leaves a partial destination and reports DECB_ERR_INTERRUPTED.

decbfile.h · `-ldecbfile`

decb_createRecordFile

```
byte decb_createRecordFile(DECBRecordFile *recFile, byte driveNo, const char *filename, word headerSize, word recordSize);
```

Choose a header length in bytes and a fixed nonzero record size. Use the same layout when reopening. The caller owns the DECBRecordFile state.

decbfile.h · `-ldecbfile`

decb_createSectorFile

```
byte decb_createSectorFile(DECBFile *file, byte driveNo, const char *filename, byte fileType, byte fileFormat);
```

Create a new empty file and supply the directory type and encoding as DECB_TYPE_* and DECB_FORMAT_*. An existing name can return DECB_ERR_ALREADY_EXISTS. Work on writable practice media.

decbfile.h · `-ldecbfile`

decb_denormalizeFilename

```
void decb_denormalizeFilename(char dest[13], char src[11], byte lowerCase);
```

The on-disk field is eight name characters and three extension characters padded with spaces. For normalization supply 12 destination bytes; for the dotted result allow 13. The latter can choose lowercase output.

decbfile.h · `-ldecbfile`

decb_dskcon

```
byte decb_dskcon(byte operation, void *buffer, byte drive, byte track, byte sector);
```

Raw DSKCON operation: supply a 256-byte buffer, track 0–34 and physical sector 1–18. This call uses nonzero for success, the opposite convention from decb_readSector. Raw writes can alter the directory or allocation table.

decbfile.h · `-ldecbfile`

DECB_DSKCON_READ

```
DECB_DSKCON_READ = 2
```

Select the direction of a raw physical sector transfer. Work on a private disk image when experimenting with writes.

decbfile.h · `-ldecbfile`

DECB_DSKCON_WRITE

```
DECB_DSKCON_WRITE = 3
```

Select the direction of a raw physical sector transfer. Work on a private disk image when experimenting with writes.

decbfile.h · `-ldecbfile`

DECB_ERR_ALREADY_EXISTS

```
DECB_ERR_ALREADY_EXISTS = 8
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.

decbfile.h · `-ldecbfile`

DECB_ERR_CORRUPT_DIR

```
DECB_ERR_CORRUPT_DIR = 7
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.

decbfile.h · `-ldecbfile`

DECB_ERR_CORRUPT_FAT

```
DECB_ERR_CORRUPT_FAT = 6
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.

decbfile.h · `-ldecbfile`

DECB_ERR_DRIVE_NOT_REGISTERED

```
DECB_ERR_DRIVE_NOT_REGISTERED = 11
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_END_OF_DIR

```
DECB_ERR_END_OF_DIR = 10
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_FAT_NOT_LOADED

```
DECB_ERR_FAT_NOT_LOADED = 5
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_INTERRUPTED

```
DECB_ERR_INTERRUPTED = 9
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_INVALID_ARGUMENT

```
DECB_ERR_INVALID_ARGUMENT = 1
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_IO

```
DECB_ERR_IO = 2
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_NOT_FOUND

```
DECB_ERR_NOT_FOUND = 3
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

DECB_ERR_OUT_OF_SPACE

```
DECB_ERR_OUT_OF_SPACE = 4
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.
decbfile.h · `-ldecbfile`

decb_findDirEntry

```
byte decb_findDirEntry(byte forNewFile, byte driveNo, char normalizedFilename[12], byte dirSectorBuffer[256], byte *dirSectorNum, byte *byteOffsetInDirSector);
```

Supply a normalized 12-byte name and 256-byte scratch sector. Select lookup or new-file search. Check the returned status before using the directory sector and offset outputs.

decbfile.h · `-ldecbfile`

decb_findFreeGranule

```
byte decb_findFreeGranule(byte driveNo);
```

Search the drive allocation table for unused space. A result of 0xFF means a free entry could not be obtained.

decbfile.h · `-ldecbfile`

decb_flush

```
byte decb_flush(DECBFile *file, DECBSeqWriteBuffer *writeBuffer);
```

Check whether buffered data fills a sector. Flush to the open file and check the returned status. Finish the last partial sector and close the file.

decbfile.h · `-ldecbfile`

DECB_FORMAT_ASCII

```
DECB_FORMAT_ASCII = 255
```

Directory metadata for decb_createSectorFile. A file has both a type and an encoding.

decbfile.h · `-ldecbfile`

DECB_FORMAT_BINARY

```
DECB_FORMAT_BINARY = 0
```

Directory metadata for decb_createSectorFile. A file has both a type and an encoding.

decbfile.h · `-ldecbfile`

decb_freeGranuleChain

```
byte decb_freeGranuleChain(byte driveNo, byte firstGranule);
```

Release the chain beginning at the given granule in the FAT cache. Flush the FAT explicitly. Use a spare image for allocation-table work.

decbfile.h · `-ldecbfile`

decb_getFileGranuleFromIndex

```
byte decb_getFileGranuleFromIndex(DECBFile *file, byte fileGranIndex, byte *fatEntryIndex);
```

Translate a file-relative granule index using the open file's allocation chain. Within a granule the sector offset is 0–8. Check the DECB status before using the physical coordinates.

decbfile.h · `-ldecbfile`

decb_getFileSizeFromDirEntry

```
byte decb_getFileSizeFromDirEntry(byte driveNo, DECBDirEntry *entry, word *numSectors, dword
*lengthInBytes);
```

The return is a granule count, with 255 indicating failure. The separate outputs distinguish used sectors and, where requested, the actual byte length.

decbfile.h · `-ldecbfile`

decb_getFileSizeFromFilename

```
byte decb_getFileSizeFromFilename(byte driveNo, const char *filename, word *numSectors, dword
*lengthInBytes);
```

The return is a granule count, with 255 indicating failure. The separate outputs distinguish used sectors and, where requested, the actual byte length.

decbfile.h · `-ldecbfile`

decb_getFreeSpaceAddress

```
byte *decb_getFreeSpaceAddress(DECBSeqWriteBuffer *writeBuffer);
```

Get the write buffer's next free position and remaining capacity. Only after copying bytes into that space, report the count you actually wrote.
decbfile.h · `-ldecbfile`

decb_getFreeSpaceSize

```
word decb_getFreeSpaceSize(DECBSeqWriteBuffer *writeBuffer);
```

Get the write buffer's next free position and remaining capacity. Only after copying bytes into that space, report the count you actually wrote.
decbfile.h · `-ldecbfile`

decb_getNumFreeGranules

```
byte decb_getNumFreeGranules(byte driveNo, byte *numFreeGranules);
```

On DECB_OK the output byte holds the number of free 2,304-byte allocation units on this drive. The return itself is a status.
decbfile.h · `-ldecbfile`

decb_getNumGranulesInOpenFile

```
byte decb_getNumGranulesInOpenFile(DECBFile *file, byte *lastFileGranule, word *numSectors);
```

The return is a granule count, with 255 indicating failure. The separate outputs distinguish used sectors and, where requested, the actual byte length.
decbfile.h · `-ldecbfile`

decb_getNumRecords

```
word decb_getNumRecords(DECBRecordFile *recFile);
```

Read the count from the open record-file state. Valid existing record indices start at zero and end before that count.
decbfile.h · `-ldecbfile`

decb_getOpenFileSize

```
byte decb_getOpenFileSize(DECBFile *file, word *numSectors, dword *lengthInBytes);
```

The return is a granule count, with 255 indicating failure. The separate outputs distinguish used sectors and, where requested, the actual byte length.
decbfile.h · `-ldecbfile`

decb_hasFullSector

```
byte decb_hasFullSector(DECBSeqWriteBuffer *writeBuffer);
```

Check whether buffered data fills a sector. Flush to the open file and check the returned status. Finish the last partial sector and close the file.
decbfile.h · `-ldecbfile`

decb_init

```
byte decb_init(DECBDrive *driveArray, word numDrives);
```

Give the library an array with one DECBDrive for each drive you want open at the same time. Keep the array allocated through shutdown. A status of DECB_OK permits the next step, registration.
decbfile.h · `-ldecbfile` · See it used in disk.c

decb_initSeqWriteBuffer

```
void decb_initSeqWriteBuffer(DECBSeqWriteBuffer *writeBuffer);
```

Prepare the caller's buffer for sequential output to a newly created file. Keep it alive until its contents have been flushed.
decbfile.h · `-ldecbfile`

decb_isValidFATEntry

```
byte decb_isValidFATEntry(byte entry);
```

Test whether a byte has a legal allocation-table encoding. This alone cannot validate the chain of an entire file.

decbfile.h · `-ldecbfile`

decb_kill

```
byte decb_kill(byte driveNo, const char *filename);
```

Delete a named file or change its directory name on a registered drive. Check the DECB status and use a working copy when experimenting.

decbfile.h · `-ldecbfile`

DECB_LAST_DIR_SECTOR

```
DECB_LAST_DIR_SECTOR = 18
```

These values describe the library's ordinary 35-track Disk BASIC floppy. A granule spans nine 256-byte sectors; the directory has its own reserved track.

decbfile.h · `-ldecbfile`

DECB_MAX_NUM_GRANULES

```
DECB_MAX_NUM_GRANULES = 68
```

These values describe the library's ordinary 35-track Disk BASIC floppy. A granule spans nine 256-byte sectors; the directory has its own reserved track.

decbfile.h · `-ldecbfile`

DECB_MAX_NUM_TRACKS

```
DECB_MAX_NUM_TRACKS = 35
```

These values describe the library's ordinary 35-track Disk BASIC floppy. A granule spans nine 256-byte sectors; the directory has its own reserved track.

decbfile.h · `-ldecbfile`

decb_normalizeFilename

```
void decb_normalizeFilename(char dest[12], const char *src);
```

The on-disk field is eight name characters and three extension characters padded with spaces. For normalization supply 12 destination bytes; for the dotted result allow 13. The latter can choose lowercase output.

decbfile.h · `-ldecbfile`

DECB_OK

```
DECB_OK = 0
```

A file-library status code. Compare the returned status before using an open file or the contents of a sector buffer.

decbfile.h · `-ldecbfile` · See it used in disk.c

decb_openDir

```
byte decb_openDir(byte driveNo, DECBDirIterator *dirIter);
```

Keep the iterator for the duration of a directory scan. Each successful read yields a pointer to an entry in its buffer. Copy an entry before advancing if you need it later. Stop on DECB_ERR_END_OF_DIR or another error, then close.

decbfile.h · `-ldecbfile`

decb_openRecordFile

```
byte decb_openRecordFile(DECBRecordFile *recFile, byte driveNo, const char *filename, word
headerSize, word recordSize);
```

Choose a header length in bytes and a fixed nonzero record size. Use the same layout when reopening. The caller owns the DECBRecordFile state.
decbfile.h · `-ldecbfile`

decb_openSectorFile

```
byte decb_openSectorFile(DECBFile *file, byte driveNo, const char *filename);
```

Open a dotted name such as WORDS.DAT on a registered drive. Supply your own DECBFile object. At status 0 it describes an open file; status 3 means the name was missing. Close only after a successful open.
decbfile.h · `-ldecbfile` · See it used in disk.c

decb_openSectorFileFromDirEntry

```
byte decb_openSectorFileFromDirEntry(DECBFile *file, byte driveNo, DECBDirEntry *entry);
```

Open using a directory record from the same registered drive. Keep your own DECBFile until close and inspect status before reading.
decbfile.h · `-ldecbfile`

decb_readDir

```
byte decb_readDir(DECBDirIterator *dirIter, DECBDirEntry **dirEntry);
```

Keep the iterator for the duration of a directory scan. Each successful read yields a pointer to an entry in its buffer. Copy an entry before advancing if you need it later. Stop on DECB_ERR_END_OF_DIR or another error, then close.
decbfile.h · `-ldecbfile`

decb_readFAT

```
byte decb_readFAT(byte driveNo);
```

Load a registered drive's allocation table before working on it. Change entries only through the setter, then write the table back. These calls can change what data every file on the image owns.
decbfile.h · `-ldecbfile`

decb_readHeader

```
byte decb_readHeader(DECBRecordFile *recFile, byte *header);
```

Provide storage covering the file's entire configured header length. These calls transfer that fixed-length binary block.
decbfile.h · `-ldecbfile`

decb_readRecord

```
byte decb_readRecord(DECBRecordFile *recFile, byte *record, word recordIndex);
```

Supply a complete record-sized buffer and a zero-based record index. Reading past the end can return DECB_ERR_NOT_FOUND. Writing past the end extends the file.
decbfile.h · `-ldecbfile`

decb_readSector

```
byte decb_readSector(DECBFile *file, byte sectorBuffer[256], word fileSectorIndex);
```

To read a disk sector **inside a file**, open the file first. Index 0 is its first sector. Reserve 256 writable bytes, even for a short final sector. Status 0 means the sector was read; DECB_ERR_NOT_FOUND can mean the index is past the file. A successful read does not add a string terminator.
decbfile.h · `-ldecbfile` · See it used in disk.c · Follow the read in chapter 6

decb_registerDrive

```
byte decb_registerDrive(byte decbDriveNo);
```

Associate a BASIC drive number with an available state record before opening files. A second registration may report DECB_ERR_ALREADY_EXISTS. Finish using its files before releasing the record.
decbfile.h · `-ldecbfile` · See it used in disk.c

decb_registerWrittenBytes

```
void decb_registerWrittenBytes(DECBSeqWriteBuffer *writeBuffer, word numBytes);
```

Get the write buffer's next free position and remaining capacity. Only after copying bytes into that space, report the count you actually wrote.
decbfile.h · `-ldecbfile`

decb_rename

```
byte decb_rename(byte driveNo, const char *existingFilename, const char *newFilename);
```

Delete a named file or change its directory name on a registered drive. Check the DECB status and use a working copy when experimenting.
decbfile.h · `-ldecbfile`

DECB_SEQ_WRITE_BUFFER_SIZE

```
DECB_SEQ_WRITE_BUFFER_SIZE = 512
```

Capacity in bytes of the data array inside DECBSeqWriteBuffer. Query the free-space size before copying into a partly occupied buffer.
decbfile.h · `-ldecbfile`

decb_setDskConAddresses

```
byte decb_setDskConAddresses(void (*dskconRoutine)(), DECBDskConVariables *vars);
```

Redirect the disk routines to a different DSKCON implementation and its matching contiguous variables. The standard ROM-backed path needs no override. Initialize the replacement and keep its IRQ and NMI services operative.
decbfile.h · `-ldecbfile`

decb_setFATEntry

```
byte decb_setFATEntry(byte driveNo, byte entryIndex, byte newValue);
```

Load a registered drive's allocation table before working on it. Change entries only through the setter, then write the table back. These calls can change what data every file on the image owns.
decbfile.h · `-ldecbfile`

decb_setNumBytesUsedInLastSector

```
byte decb_setNumBytesUsedInLastSector(DECBFile *file, word numBytesUsedInLastSector);
```

Record the logical byte count in the last sector, 0–256. Close the file to finish updating the directory.
decbfile.h · `-ldecbfile`

decb_shutdown

```
byte decb_shutdown();
```

After closing files, release the library's drive registrations. If you explicitly changed the FAT, write it before shutdown. Keep the drive array valid until this call returns.
decbfile.h · `-ldecbfile` · See it used in disk.c

decb_truncateOpenFile

```
byte decb_truncateOpenFile(DECBFile *file, word newNumSectors, word numBytesUsedInLastSector);
```

Adjust the sector length and final byte count within the existing allocation. Close the file after success so the new size reaches disk.
decbfile.h · `-ldecbfile`

decb_truncateOpenRecordFile

```
byte decb_truncateOpenRecordFile(DECBRecordFile *recFile, word recordIndex);
```

Truncate at a record boundary, then close the open record file to complete updates. Check each status independently.

decbfile.h · `-ldecbfile`

DECB_TYPE_ASCII_TEXT

```
DECB_TYPE_ASCII_TEXT = 3
```

Directory metadata for decb_createSectorFile. A file has both a type and an encoding.

decbfile.h · `-ldecbfile`

DECB_TYPE_BASIC_DATA

```
DECB_TYPE_BASIC_DATA = 1
```

Directory metadata for decb_createSectorFile. A file has both a type and an encoding.

decbfile.h · `-ldecbfile`

DECB_TYPE_BASIC_PROGRAM

```
DECB_TYPE_BASIC_PROGRAM = 0
```

Directory metadata for decb_createSectorFile. A file has both a type and an encoding.

decbfile.h · `-ldecbfile`

DECB_TYPE_MACHINE_CODE

```
DECB_TYPE_MACHINE_CODE = 2
```

Directory metadata for decb_createSectorFile. A file has both a type and an encoding.

decbfile.h · `-ldecbfile`

decb_unregisterAllDrives

```
byte decb_unregisterAllDrives();
```

Associate a BASIC drive number with an available state record before opening files. A second registration may report DECB_ERR_ALREADY_EXISTS. Finish using its files before releasing the record.

decbfile.h · `-ldecbfile`

decb_unregisterDrive

```
byte decb_unregisterDrive(byte decbDriveNo);
```

Associate a BASIC drive number with an available state record before opening files. A second registration may report DECB_ERR_ALREADY_EXISTS. Finish using its files before releasing the record.

decbfile.h · `-ldecbfile`

decb_writeFAT

```
byte decb_writeFAT(byte driveNo);
```

Load a registered drive's allocation table before working on it. Change entries only through the setter, then write the table back. These calls can change what data every file on the image owns.

decbfile.h · `-ldecbfile`

decb_writeHeader

```
byte decb_writeHeader(DECBRecordFile *recFile, byte *header);
```

Provide storage covering the file's entire configured header length. These calls transfer that fixed-length binary block.

decbfile.h · `-ldecbfile`

decb_writeRecord

```
byte decb_writeRecord(DECBRecordFile *recFile, byte *record, word recordIndex);
```

Supply a complete record-sized buffer and a zero-based record index. Reading past the end can return DECB_ERR_NOT_FOUND. Writing past the end extends the file.

decbfile.h · `-ldecbfile`

decb_writeSector

```
byte decb_writeSector(DECBFile *file, const byte sectorBuffer[256], word fileSectorIndex);
```

Write a full 256-byte buffer at a zero-based logical sector of the open file. The result is a DECB status. Set the last-sector byte count when the useful data ends partway through it.

decbfile.h · `-ldecbfile`

DECBDirEntry

```
typedef struct DECBDirEntry { … } DECBDirEntry;
```

Keep the iterator for the duration of a directory scan. Each successful read yields a pointer to an entry in its buffer. Copy an entry before advancing if you need it later. Stop on DECB_ERR_END_OF_DIR or another error, then close.

decbfile.h · `-ldecbfile`

DECBDirIterator

```
typedef struct DECBDirIterator { … } DECBDirIterator;
```

Keep the iterator for the duration of a directory scan. Each successful read yields a pointer to an entry in its buffer. Copy an entry before advancing if you need it later. Stop on DECB_ERR_END_OF_DIR or another error, then close.

decbfile.h · `-ldecbfile`

DECBDrive

```
typedef struct DECBDrive { … } DECBDrive;
```

Give the library an array with one DECBDrive for each drive you want open at the same time. Keep the array allocated through shutdown. A status of DECB_OK permits the next step, registration.

decbfile.h · `-ldecbfile` · See it used in disk.c

DECBDskConVariables

```
typedef struct DECBDskConVariables { … } DECBDskConVariables;
```

Redirect the disk routines to a different DSKCON implementation and its matching contiguous variables. The standard ROM-backed path needs no override. Initialize the replacement and keep its IRQ and NMI services operative.

decbfile.h · `-ldecbfile`

DECBFile

```
typedef struct DECBFile { … } DECBFile;
```

Open a dotted name such as WORDS.DAT on a registered drive. Supply your own DECBFile object. At status 0 it describes an open file; status 3 means the name was missing. Close only after a successful open.

decbfile.h · `-ldecbfile` · See it used in disk.c

DECBRecordFile

```
typedef struct DECBRecordFile { … } DECBRecordFile;
```

Choose a header length in bytes and a fixed nonzero record size. Use the same layout when reopening. The caller owns the DECBRecordFile state.

decbfile.h · `-ldecbfile`

DECBSeqWriteBuffer

`typedef struct DECBSeqWriteBuffer { … } DECBSeqWriteBuffer;`

Prepare the caller's buffer for sequential output to a newly created file. Keep it alive until its contents have been flushed.

decbfile.h · `-ldecbfile`

dskcon-standalone.h

Operate the floppy controller while the Disk BASIC ROM is banked out. This optional driver must be present in the installed toolchain. Link decbfile then dskcon-standalone when replacing the disk API's ROM backend.

DCBPT · DCDRV · DCOPC · DCSEC · DCSTA · DCTRK · DNMIVC · DR0TRK · DRGRAM · dskcon_driveEnableMasks · dskcon_init · dskcon_irqService · dskcon_nmiService · dskcon_NmiServiceFunctionPointer · dskcon_processSector · dskcon_shutdown · DSKREG · FDCREG · NMIFLG · RDYTMR

DCBPT

`extern unsigned char *DCBPT;`

The standalone driver's physical sector request uses a drive (0–3), track (0–34), sector (1–18) and pointer to 256 bytes. Request 2 reads and 3 writes. Read the status after the transfer.

dskcon-standalone.h · `-ldskcon-standalone`

DCDRV

`extern unsigned char DCDRV;`

The standalone driver's physical sector request uses a drive (0–3), track (0–34), sector (1–18) and pointer to 256 bytes. Request 2 reads and 3 writes. Read the status after the transfer.

dskcon-standalone.h · `-ldskcon-standalone`

DCOPC

`extern unsigned char DCOPC;`

The standalone driver's physical sector request uses a drive (0–3), track (0–34), sector (1–18) and pointer to 256 bytes. Request 2 reads and 3 writes. Read the status after the transfer.

dskcon-standalone.h · `-ldskcon-standalone`

DCSEC

`extern unsigned char DCSEC;`

The standalone driver's physical sector request uses a drive (0–3), track (0–34), sector (1–18) and pointer to 256 bytes. Request 2 reads and 3 writes. Read the status after the transfer.

dskcon-standalone.h · `-ldskcon-standalone`

DCSTA

`extern unsigned char DCSTA;`

The standalone driver's physical sector request uses a drive (0–3), track (0–34), sector (1–18) and pointer to 256 bytes. Request 2 reads and 3 writes. Read the status after the transfer.

dskcon-standalone.h · `-ldskcon-standalone`

DCTRK

`extern unsigned char DCTRK;`

The standalone driver's physical sector request uses a drive (0–3), track (0–34), sector (1–18) and pointer to 256 bytes. Request 2 reads and 3 writes. Read the status after the transfer.

dskcon-standalone.h · `-ldskcon-standalone`

DNMIVC

```
extern void *DNMIVC;
```

Driver state for the motor, selected drives and interrupt handling. Let dskcon_init, the service routines and dskcon_shutdown maintain it during ordinary file I/O.

dskcon-standalone.h · `-ldskcon-standalone`

DR0TRK

```
extern unsigned char DR0TRK[4];
```

Driver state for the motor, selected drives and interrupt handling. Let dskcon_init, the service routines and dskcon_shutdown maintain it during ordinary file I/O.

dskcon-standalone.h · `-ldskcon-standalone`

DRGRAM

```
extern unsigned char DRGRAM;
```

Driver state for the motor, selected drives and interrupt handling. Let dskcon_init, the service routines and dskcon_shutdown maintain it during ordinary file I/O.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_driveEnableMasks

```
extern unsigned char dskcon_driveEnableMasks[4];
```

Driver state for the motor, selected drives and interrupt handling. Let dskcon_init, the service routines and dskcon_shutdown maintain it during ordinary file I/O.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_init

```
unsigned long dskcon_init(dskcon_NmiServiceFunctionPointer newNMIService);
```

This optional standalone FD-502 driver takes responsibility for controller interrupts. With interrupts masked, install the NMI handler and save the exact initialization return value for shutdown. Service the driver's IRQ from a suitable periodic guest ISR, then restore the vectors on exit.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_irqService

```
void dskcon_irqService();
```

This optional standalone FD-502 driver takes responsibility for controller interrupts. With interrupts masked, install the NMI handler and save the exact initialization return value for shutdown. Service the driver's IRQ from a suitable periodic guest ISR, then restore the vectors on exit.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_nmiService

```
interrupt void dskcon_nmiService();
```

This optional standalone FD-502 driver takes responsibility for controller interrupts. With interrupts masked, install the NMI handler and save the exact initialization return value for shutdown. Service the driver's IRQ from a suitable periodic guest ISR, then restore the vectors on exit.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_NmiServiceFunctionPointer

```
typedef interrupt void (*dskcon_NmiServiceFunctionPointer)();
```

A callback type. Implement the shown return and argument types exactly. Keep any supplied user data alive while the library can call back.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_processSector

```
void dskcon_processSector();
```

Fill the driver's operation, drive, track, sector and buffer variables, then call. DCSTA is the resulting hardware error mask; zero signals success. A transfer uses 256 bytes.

dskcon-standalone.h · `-ldskcon-standalone`

dskcon_shutdown

```
void dskcon_shutdown(unsigned long initReturnValue);
```

This optional standalone FD-502 driver takes responsibility for controller interrupts. With interrupts masked, install the NMI handler and save the exact initialization return value for shutdown. Service the driver's IRQ from a suitable periodic guest ISR, then restore the vectors on exit.

dskcon-standalone.h · `-ldskcon-standalone`

DSKREG

```
DSKREG = 65344
```

Guest addresses of the drive-control and floppy-controller registers respectively. Ordinary clients use dskcon_processSector or a file-level API.

dskcon-standalone.h · `-ldskcon-standalone`

FDCREG

```
FDCREG = 65352
```

Guest addresses of the drive-control and floppy-controller registers respectively. Ordinary clients use dskcon_processSector or a file-level API.

dskcon-standalone.h · `-ldskcon-standalone`

NMIFLG

```
extern unsigned char NMIFLG;
```

Driver state for the motor, selected drives and interrupt handling. Let dskcon_init, the service routines and dskcon_shutdown maintain it during ordinary file I/O.

dskcon-standalone.h · `-ldskcon-standalone`

RDYTMR

```
extern unsigned char RDYTMR;
```

Driver state for the motor, selected drives and interrupt handling. Let dskcon_init, the service routines and dskcon_shutdown maintain it during ordinary file I/O.

dskcon-standalone.h · `-ldskcon-standalone`

vcc/ssc.h

Add **ssc** in Project Settings → Libraries and select the Speech/Sound Cartridge in the machine configuration. Supply its two firmware files. Use one command producer at a time. Before sending a command, select cartridge sound with ssc_route_audio. Give the sound time to finish before restoring the prior audio route.

ssc_status · ssc_try_write · ssc_route_audio · ssc_restore_audio · ssc_reset · ssc_phrase · ssc_begin_phrase · ssc_service · ssc_say · ssc_play_effect · ssc_play_noise · SSC_OK · SSC_TIMEOUT · SSC_INVALID · SSC_PENDING · SSC_READY · SSC_SPEECH_IDLE · SSC_EFFECT_IDLE · SSC_MAX_TEXT

The cartridge has eight speech slots and eight independent sound slots. Loading a slot is silent. Playback is a separate command. Each call below has a per-byte polling limit. A zero limit gives a timeout on a valid call. If a transfer times out partway through, reset and reload your slots before sending another command. The slots are lost when the cartridge resets.

- **Store and speak.** ssc_store_phrase, ssc_play_phrase, ssc_store_allophones, ssc_play_allophones.
- **Store and play sound.** ssc_store_enveloped_tone, ssc_play_stored_effect.

- **Envelope shapes.** SSC_ENV_DECAY, SSC_ENV_REPEAT_DECAY, SSC_ENV_REPEAT_RISE, SSC_ENV_TRIANGLE.
- **Pauses.** PA1, PA2, PA3, PA4, PA5.
- **Vowels and syllables.** OY, AY, EH, IH, AX, IY, EY, UW1, AO, AA, AE, UH, UW2, AW, XR, ER1, ER2, OW, OR, AR, YR, EL.
- **Consonants.** BB1, BB2, CH, DD1, DD2, DH1, DH2, FF, GG1, GG2, GG3, HH1, HH2, JH, KK1, KK2, KK3, LL, MM, NG, NN1, NN2, PP, RR1, RR2, SH, SS, TH, TT1, TT2, VV, WH, WW, YY1, YY2, ZH, ZZ.

ssc_status

```
unsigned char ssc_status(void);
```

Read the cartridge status flags. Test READY before each command byte. Idle flags can take time to change after submission. An unmapped read does not reliably identify whether the cartridge is installed.

vcc/ssc.h

ssc_try_write

```
unsigned char ssc_try_write(unsigned char value);
```

Attempt one command byte. Returns 1 when the cartridge accepted it and 0 while it is busy. On 0, keep the same byte for your next attempt.

vcc/ssc.h

ssc_route_audio

```
unsigned int ssc_route_audio(void);
```

Select cartridge audio and enable the speaker. Keep the returned token until playback has finished. Joystick or cassette routines can change the shared CoCo audio multiplexer.

vcc/ssc.h

ssc_restore_audio

```
void ssc_restore_audio(unsigned int saved);
```

Pass the original route token after playback to restore the CoCo's previous audio routing. This call returns immediately.

vcc/ssc.h

ssc_reset

```
void ssc_reset(void);
```

Reset the cartridge and discard pending speech, effects and buffered commands. Use deliberately when recovering from a partial submission.

vcc/ssc.h

ssc_phrase

```
typedef struct { … } ssc_phrase;
```

Keep this state and its input string alive from ssc_begin_phrase until the last ssc_service. Give a phrase exclusive use of the command port.

vcc/ssc.h

ssc_begin_phrase

```
unsigned char ssc_begin_phrase(ssc_phrase *phrase, const char *text);
```

Validate 1–62 printable ASCII characters and initialize the phrase state. This call sends no bytes. Keep the original text unchanged until submission finishes.

vcc/ssc.h

ssc_service

```
unsigned char ssc_service(ssc_phrase *phrase);
```

Try at most one byte from a prepared phrase. Continue the main loop on SSC_PENDING. SSC_OK means the entire command has been submitted; playback may still be in progress.
vcc/ssc.h

ssc_say

```
unsigned char ssc_say(const char *text, unsigned int poll_limit);
```

Submit a printable phrase synchronously. The poll limit applies separately to each byte. Timeout may leave a partial command; reset before sending something unrelated. Successful submission can return before you hear the end of the phrase.
vcc/ssc.h

ssc_play_effect

```
unsigned char ssc_play_effect(unsigned char channel, unsigned int period, unsigned char volume,
unsigned char duration, unsigned int poll_limit);
```

Submit a tone effect on channel A, B or C (0–2). Set its tone period to 1–4095 and volume to 0–15. Duration is 0–255 cartridge timer units. It replaces cartridge effect buffer 0 and ends the effect in silence.
vcc/ssc.h

ssc_play_noise

```
unsigned char ssc_play_noise(unsigned char channel, unsigned char period, unsigned char volume,
unsigned char duration, unsigned int poll_limit);
```

Submit a noise effect on channel 0–2, with noise period 0–31, volume 0–15 and duration 0–255 cartridge timer units. It shares effect buffer 0 with tone effects.
vcc/ssc.h

SSC_OK

```
SSC_OK = 0
```

The command was submitted successfully. The firmware may still be playing it.
vcc/ssc.h

SSC_TIMEOUT

```
SSC_TIMEOUT = 1
```

A bounded wait expired. Some earlier bytes may have been accepted.
vcc/ssc.h

SSC_INVALID

```
SSC_INVALID = 2
```

An argument fell outside the accepted range.
vcc/ssc.h

SSC_PENDING

```
SSC_PENDING = 3
```

The phrase still has bytes to send. Call ssc_service again.
vcc/ssc.h

SSC_READY

```
SSC_READY = 0x80
```

Mask the status byte with this bit to check whether the cartridge accepts another byte.
vcc/ssc.h

SSC_SPEECH_IDLE

`SSC_SPEECH_IDLE = 0x40`

Set when the speech synthesizer is idle. Allow time for a submitted phrase to start before waiting for completion.

vcc/ssc.h

SSC_EFFECT_IDLE

`SSC_EFFECT_IDLE = 0x20`

Set when the sound effect is idle. It reports the effect engine independently of speech.

vcc/ssc.h

SSC_MAX_TEXT

`SSC_MAX_TEXT = 62`

Maximum number of printable ASCII characters in a phrase; leave space in the C string for its terminator.

vcc/ssc.h

ssc_store_phrase

```
unsigned char ssc_store_phrase(unsigned char slot, const char *text, unsigned int poll_limit);
```

Save 1–62 printable ASCII characters in speech slot 0–7. Lowercase letters are converted to uppercase. The slot can be played again after the caller reuses the string. Replaces earlier text or allophones in the same slot.

vcc/ssc.h

ssc_play_phrase

```
unsigned char ssc_play_phrase(unsigned char slot, unsigned int poll_limit);
```

Speak the English text most recently saved in this speech slot. Store it first with ssc_store_phrase. Wait for it to finish before replacing its contents.

vcc/ssc.h

ssc_store_allophones

```
unsigned char ssc_store_allophones(unsigned char slot, const unsigned char *codes, unsigned int count, unsigned int poll_limit);
```

Save 1–62 SP0256 sound codes from a byte array in speech slot 0–7. Pass the number of sounds, including any pauses you put between them. The helper supplies a final quiet pause and the end marker. Invalid input leaves the old slot intact.

vcc/ssc.h

ssc_play_allophones

```
unsigned char ssc_play_allophones(unsigned char slot, unsigned int poll_limit);
```

Replay a custom pronunciation saved with ssc_store_allophones. Select the cartridge audio route before listening. The loop in Mission Announcer accepts a joystick press during playback.

vcc/ssc.h

ssc_store_enveloped_tone

```
unsigned char ssc_store_enveloped_tone(unsigned char slot, unsigned char channel, unsigned int tone_period, unsigned char shape, unsigned int envelope_period, unsigned char duration, unsigned int poll_limit);
```

Save a fading or repeating tone in sound slot 0–7. Channels A, B and C are 0–2. A larger tone period (1–4095) lowers the pitch. Shape (0–15) chooses the volume pattern. A larger envelope period (1–65535) slows that pattern. Duration (0–255) counts cartridge timer units before silencing the chosen channel. The three channels share one envelope generator.

vcc/ssc.h

ssc_play_stored_effect

```
unsigned char ssc_play_stored_effect(unsigned char slot, unsigned int poll_limit);
```

Start the sequence in a previously loaded sound slot. This bank is separate from speech. The fixed-volume tone and noise helpers replace sound slot 0, so choose another slot to keep a custom alert across those calls.

vcc/ssc.h

SSC_MAX_ALLOPHONES

```
SSC_MAX_ALLOPHONES = 62
```

Maximum caller-supplied sound codes in one stored pronunciation. Allocate an unsigned char for each code, including pauses.

vcc/ssc.h

SSC_ENV_DECAY

```
SSC_ENV_DECAY = 0
```

Start at full volume, fall to silence and stay there.

vcc/ssc.h

SSC_ENV_REPEAT_DECAY

```
SSC_ENV_REPEAT_DECAY = 8
```

Start loud, fade, then jump back to full volume for each repetition.

vcc/ssc.h

SSC_ENV_REPEAT_RISE

```
SSC_ENV_REPEAT_RISE = 12
```

Start quiet and swell, then drop back to silence for each repetition.

vcc/ssc.h

SSC_ENV_TRIANGLE

```
SSC_ENV_TRIANGLE = 14
```

Rise and fall repeatedly between silence and full volume.

vcc/ssc.h

SSC_AL_PA1

```
SSC_AL_PA1 = 0
```

A very short silent gap; the store helper also appends this pause at the end.

vcc/ssc.h

SSC_AL_PA2

```
SSC_AL_PA2 = 1
```

A short silent gap.

vcc/ssc.h

SSC_AL_PA3

```
SSC_AL_PA3 = 2
```

A medium silent gap.

vcc/ssc.h

SSC_AL_PA4

```
SSC_AL_PA4 = 3
```

A longer silent gap.

vcc/ssc.h

SSC_AL_PA5

`SSC_AL_PA5 = 4`

A long silent gap.

vcc/ssc.h

SSC_AL_OY

`SSC_AL_OY = 5`

Listen for the vowel in *voice*.

vcc/ssc.h

SSC_AL_AY

`SSC_AL_AY = 6`

Listen for the vowel in *flight*.

vcc/ssc.h

SSC_AL_EH

`SSC_AL_EH = 7`

Listen for the vowel in *jet*.

vcc/ssc.h

SSC_AL_KK3

`SSC_AL_KK3 = 8`

Listen for the initial k sound in *cold*.

vcc/ssc.h

SSC_AL_PP

`SSC_AL_PP = 9`

Listen for the initial p in *pilot*.

vcc/ssc.h

SSC_AL_JH

`SSC_AL_JH = 10`

Listen for the initial j in *jet*.

vcc/ssc.h

SSC_AL_NN1

`SSC_AL_NN1 = 11`

Listen for the final n in *sun*.

vcc/ssc.h

SSC_AL_IH

`SSC_AL_IH = 12`

Listen for the vowel in *ship*.

vcc/ssc.h

SSC_AL_TT2

`SSC_AL_TT2 = 13`

Listen for the initial t in *two*.

vcc/ssc.h

SSC_AL_RR1

`SSC_AL_RR1 = 14`

Listen for the initial r in *red*.

vcc/ssc.h

SSC_AL_AX

`SSC_AL_AX = 15`

Listen for the weak first vowel in *about*.

vcc/ssc.h

SSC_AL_MM

`SSC_AL_MM = 16`

Listen for the initial m in *moon*.

vcc/ssc.h

SSC_AL_TT1

`SSC_AL_TT1 = 17`

Listen for the final t in *light*.

vcc/ssc.h

SSC_AL_DH1

`SSC_AL_DH1 = 18`

Listen for the voiced th in *this*.

vcc/ssc.h

SSC_AL_IY

`SSC_AL_IY = 19`

Listen for the vowel in *beam*.

vcc/ssc.h

SSC_AL_EY

`SSC_AL_EY = 20`

Listen for the vowel in *space*.

vcc/ssc.h

SSC_AL_DD1

`SSC_AL_DD1 = 21`

Listen for the final d in *land*.

vcc/ssc.h

SSC_AL_UW1

`SSC_AL_UW1 = 22`

Listen for the oo vowel in *too*.

vcc/ssc.h

SSC_AL_AO

`SSC_AL_AO = 23`

Listen for the aw vowel in *launch*.

vcc/ssc.h

SSC_AL_AA

`SSC_AL_AA = 24`

Listen for the ah vowel in *pod*.

vcc/ssc.h

SSC_AL_YY2

`SSC_AL_YY2 = 25`

Listen for the initial y in *yard*.

vcc/ssc.h

SSC_AL_AE

`SSC_AL_AE = 26`

Listen for the vowel in *map*.

vcc/ssc.h

SSC_AL_HH1

`SSC_AL_HH1 = 27`

Listen for the initial h in *heat*.

vcc/ssc.h

SSC_AL_BB1

`SSC_AL_BB1 = 28`

Listen for the initial b in *base*.

vcc/ssc.h

SSC_AL_TH

`SSC_AL_TH = 29`

Listen for the unvoiced th in *thrust*.

vcc/ssc.h

SSC_AL_UH

`SSC_AL_UH = 30`

Listen for the vowel in *push*.

vcc/ssc.h

SSC_AL_UW2

`SSC_AL_UW2 = 31`

Listen for the long oo in *moon*.

vcc/ssc.h

SSC_AL_AW

`SSC_AL_AW = 32`

Listen for the vowel in *cloud*.

vcc/ssc.h

SSC_AL_DD2

`SSC_AL_DD2 = 33`

Listen for the initial d in *dock*.

vcc/ssc.h

SSC_AL_GG3

`SSC_AL_GG3 = 34`

Listen for the final g in *flag*.

vcc/ssc.h

SSC_AL_VV

`SSC_AL_VV = 35`

Listen for the initial v in *valve*.

vcc/ssc.h

SSC_AL_GG1

`SSC_AL_GG1 = 36`

Listen for the initial g in *gone*.

vcc/ssc.h

SSC_AL_SH

`SSC_AL_SH = 37`

Listen for the sh in *ship*.

vcc/ssc.h

SSC_AL_ZH

`SSC_AL_ZH = 38`

Listen for the middle consonant in *measure*.

vcc/ssc.h

SSC_AL_RR2

`SSC_AL_RR2 = 39`

Listen for the r after b in *brake*.

vcc/ssc.h

SSC_AL_FF

`SSC_AL_FF = 40`

Listen for the initial f in *fuel*.

vcc/ssc.h

SSC_AL_KK2

`SSC_AL_KK2 = 41`

Listen for the k after s in *scan*.

vcc/ssc.h

SSC_AL_KK1

`SSC_AL_KK1 = 42`

Listen for the initial k sound in *cap*.

vcc/ssc.h

SSC_AL_ZZ

`SSC_AL_ZZ = 43`

Listen for the initial z in *zero*.

vcc/ssc.h

SSC_AL_NG

```
SSC_AL_NG = 44
```

Listen for the final ng in *wing*.

vcc/ssc.h

SSC_AL_LL

```
SSC_AL_LL = 45
```

Listen for the initial l in *launch*.

vcc/ssc.h

SSC_AL_WW

```
SSC_AL_WW = 46
```

Listen for the initial w in *wind*.

vcc/ssc.h

SSC_AL_XR

```
SSC_AL_XR = 47
```

Listen for the air sound in *flare*.

vcc/ssc.h

SSC_AL_WH

```
SSC_AL_WH = 48
```

Listen for the breathy wh in *where* in accents that distinguish it from *wear*.

vcc/ssc.h

SSC_AL_YY1

```
SSC_AL_YY1 = 49
```

Listen for the initial y in *year*.

vcc/ssc.h

SSC_AL_CH

```
SSC_AL_CH = 50
```

Listen for the ch in *chart*.

vcc/ssc.h

SSC_AL_ER1

```
SSC_AL_ER1 = 51
```

Listen for the stressed er in *turn*.

vcc/ssc.h

SSC_AL_ER2

```
SSC_AL_ER2 = 52
```

Listen for a longer er sound, as in *burn*.

vcc/ssc.h

SSC_AL_OW

```
SSC_AL_OW = 53
```

Listen for the vowel in *probe*.

vcc/ssc.h

SSC_AL_DH2

`SSC_AL_DH2 = 54`

Listen for a second voiced th sound, as in *those*.

vcc/ssc.h

SSC_AL_SS

`SSC_AL_SS = 55`

Listen for the final s sound in *space*.

vcc/ssc.h

SSC_AL_NN2

`SSC_AL_NN2 = 56`

Listen for the initial n in *north*.

vcc/ssc.h

SSC_AL_HH2

`SSC_AL_HH2 = 57`

Listen for the initial h in *home*.

vcc/ssc.h

SSC_AL_OR

`SSC_AL_OR = 58`

Listen for the or in *orbit*.

vcc/ssc.h

SSC_AL_AR

`SSC_AL_AR = 59`

Listen for the ar in *star*.

vcc/ssc.h

SSC_AL_YR

`SSC_AL_YR = 60`

Listen for the ear sound in *near*.

vcc/ssc.h

SSC_AL_GG2

`SSC_AL_GG2 = 61`

Listen for the initial g in *gear*.

vcc/ssc.h

SSC_AL_EL

`SSC_AL_EL = 62`

Listen for the final syllable in *shuttle*.

vcc/ssc.h

SSC_AL_BB2

`SSC_AL_BB2 = 63`

Listen for a second b sound, as in *boot*.

vcc/ssc.h

vcc/orch90.h

Add **orch90** in Project Settings → Libraries and assign an Orchestra-90 cartridge. Direct sample writes work without the optional Orchestra music ROM. Each byte selects a DAC level from 0 through 255. Choose the clock constant that matches the guest CPU speed.

orch90_frame · orch90_write · orch90_play · orch90_start · orch90_active · orch90_stop · ORCH90_CLOCK_NORMAL · ORCH90_CLOCK_FAST

orch90_frame

```
struct orch90_frame
```

One interleaved stereo pair of unsigned bytes. left is the left DAC level and right the right DAC level. A block of 128 frames occupies 256 bytes.

vcc/orch90.h

orch90_write

```
void orch90_write(unsigned char left, unsigned char right);
```

Latch one unsigned level on each stereo channel. Both hold their values until the next write. To finish a centered waveform, write 128 to each channel.

vcc/orch90.h

orch90_play

```
unsigned char orch90_play(const struct orch90_frame *frames, unsigned int count, unsigned char clock);
```

Send count stereo frames from guest-addressable memory at approximately 4,000 pairs per second. The call occupies the foreground while sending samples. Interrupts can delay it. An empty valid block succeeds without a write. Bad clock, missing data or an address wrap returns 0. Success cannot establish that this write-only device is installed.

vcc/orch90.h

orch90_start

```
unsigned char orch90_start(const struct orch90_frame *frames, unsigned int count, unsigned char loop, unsigned char clock);
```

Start stereo playback from the foreground. Pass a nonempty array of stereo frames, **0** for one pass or **1** to loop, and the current guest CPU speed. Keep the array mapped and unchanged until orch90_active returns 0. Its lifetime, and that of the program and stack, must cover the entire playback.

The player uses fast speed while active and takes one stereo frame every four serviced horizontal interrupts. With NTSC timing that allows about 3,930 frames per second. Leave the ordinary BASIC vertical interrupt and IRQ enabled. Playback begins within two vertical ticks once the clock settles. Long interrupt masks or reads of the joystick's PIA can delay a sample. Reserve the horizontal interrupt, CPU clock and Orchestra DACs for the player. The GIME timer and FIRQ vector remain available.

A return of **1** means the player accepted the buffer. A zero return can mean invalid arguments, playback already in progress, a replaced BASIC IRQ vector or an unavailable horizontal interrupt. The DACs are write-only, so starting can succeed with an empty cartridge slot. Steer the Stereo Flyby while a sample repeats.

vcc/orch90.h

orch90_active

```
unsigned char orch90_active(void);
```

Returns **1** while the background player owns the sample and **0** after a one-shot finishes or orch90_stop releases it. A foreground loop can use this answer to light a playback indicator. It cannot detect a cartridge.

vcc/orch90.h

orch90_stop

```
void orch90_stop(void);
```

End background playback before changing the sample, changing its memory mapping or returning to BASIC. Playback cleanup centers both DACs and releases the player's interrupt and clock settings. Calling stop again after completion is safe.

vcc/orch90.h

ORCH90_CLOCK_NORMAL

```
ORCH90_CLOCK_NORMAL = 0
```

Tell the player the guest starts at normal CPU speed. Foreground playback uses this setting for a delay. Background playback uses it to restore normal speed afterward.

vcc/orch90.h

ORCH90_CLOCK_FAST

```
ORCH90_CLOCK_FAST = 1
```

Tell the player the guest starts at fast CPU speed. Foreground playback uses the fast delay. Background playback leaves the guest at fast speed when it finishes.

vcc/orch90.h

errno.h

Use **vcccompat** and include `<errno.h>` to inspect a numeric conversion. Set errno to zero just before the call. A failure can set EINVAL or ERANGE. Disk streams have their own device error reporting.

errno

```
extern int errno;
```

The guest's numeric conversions share this error indicator. A successful conversion leaves its previous value alone. The same is true when the input contains no digits. To tell an empty conversion from a valid zero, compare the returned end pointer with the original string pointer.

errno.h · Validate a destination

EINVAL

```
EINVAL = 22
```

The requested base was outside the accepted set. Choose zero for automatic detection or a base from 2 through 36. The conversion returns zero and leaves the end pointer at the original input.

errno.h · strtol

ERANGE

```
ERANGE = 34
```

The digits describe a value outside the result type's range. The return value is the nearest signed limit for **strtol** or **ULONG_MAX** for **strtoul**. The end pointer still advances past the full sequence of valid digits.

errno.h · strtoul

ctype.h

Add **vcccompat** and include `<ctype.h>`. Pass an unsigned byte value or EOF. Cast a plain char through unsigned char first when it may contain a high-bit character. The runtime also supplies isalnum, isalpha, isdigit, isspace, tolower and toupper.

isblank · iscntrl · isgraph · islower · isprint · ispunct · isupper · isxdigit

isblank
```
int isblank(int c);
```
Test for space or horizontal tab.
ctype.h

iscntrl
```
int iscntrl(int c);
```
Test for ASCII control codes 0–31 or DEL (127).
ctype.h

isgraph
```
int isgraph(int c);
```
Test for a visible ASCII character other than space.
ctype.h

islower
```
int islower(int c);
```
Test for a lowercase ASCII letter.
ctype.h

isprint
```
int isprint(int c);
```
Test for a visible ASCII character or space.
ctype.h

ispunct
```
int ispunct(int c);
```
Test for a printable ASCII punctuation mark. Space, digits and letters return false.
ctype.h

isupper
```
int isupper(int c);
```
Test for an uppercase ASCII letter.
ctype.h

isxdigit
```
int isxdigit(int c);
```
Test for ASCII digits 0–9, A–F or a–f.
ctype.h

limits.h

Add **vcccompat** and include `<limits.h>`. A plain char is signed. Short and int occupy two bytes, long four. The target has no 64-bit integer type.
CHAR_BIT · SCHAR_MIN · SCHAR_MAX · UCHAR_MAX · CHAR_MIN · CHAR_MAX · SHRT_MIN · SHRT_MAX · USHRT_MAX · INT_MIN · INT_MAX · UINT_MAX · LONG_MIN · LONG_MAX · ULONG_MAX · MB_LEN_MAX

CHAR_BIT
```
CHAR_BIT = 8
```
Number of bits in a target byte.
limits.h

SCHAR_MIN

`SCHAR_MIN = (-127 - 1)`

Smallest representable value of signed char on the CMOC target.

limits.h

SCHAR_MAX

`SCHAR_MAX = 127`

Largest representable value of signed char on the CMOC target.

limits.h

UCHAR_MAX

`UCHAR_MAX = 255U`

Largest representable value of unsigned char on the CMOC target.

limits.h

CHAR_MIN

`CHAR_MIN = SCHAR_MIN`

Smallest representable value of plain char on the CMOC target.

limits.h

CHAR_MAX

`CHAR_MAX = SCHAR_MAX`

Largest representable value of plain char on the CMOC target.

limits.h

SHRT_MIN

`SHRT_MIN = (-32767 - 1)`

Smallest representable value of short on the CMOC target.

limits.h

SHRT_MAX

`SHRT_MAX = 32767`

Largest representable value of short on the CMOC target.

limits.h

USHRT_MAX

`USHRT_MAX = 65535U`

Largest representable value of unsigned short on the CMOC target.

limits.h

INT_MIN

`INT_MIN = SHRT_MIN`

Smallest representable value of int on the CMOC target.

limits.h

INT_MAX

`INT_MAX = SHRT_MAX`

Largest representable value of int on the CMOC target.

limits.h

UINT_MAX

`UINT_MAX = USHRT_MAX`

Largest representable value of unsigned int on the CMOC target.

limits.h

LONG_MIN

`LONG_MIN = (−2147483647L − 1L)`

Smallest representable value of long on the CMOC target.

limits.h

LONG_MAX

`LONG_MAX = 2147483647L`

Largest representable value of long on the CMOC target.

limits.h

ULONG_MAX

`ULONG_MAX = 4294967295UL`

Largest representable value of unsigned long on the CMOC target.

limits.h

MB_LEN_MAX

`MB_LEN_MAX = 1`

Maximum byte count for one character in this single-byte subset.

limits.h

stddef.h

Add **vcccompat** and include `<stddef.h>`. The runtime supplies size_t, NULL and offsetof. Only subtract pointers within the same array or one past its end.

ptrdiff_t

ptrdiff_t

```
typedef signed int ptrdiff_t;
```

Signed 16-bit difference between two positions in the same array, when representable.

stddef.h

stdint.h

Add **vcccompat** and include `<stdint.h>`. The exact-width integer types match the declarations in coco.h. Pointers occupy 16 bits. This subset also supplies the pointer-width and maximum-width types and their limits. Omit 64-bit, least-width and fast-width types when writing code for this subset.

intptr_t · uintptr_t · intmax_t · uintmax_t · INT8_MIN · INT8_MAX · UINT8_MAX · INT16_MIN · INT16_MAX · UINT16_MAX · INT32_MIN · INT32_MAX · UINT32_MAX · INTPTR_MIN · INTPTR_MAX · UINTPTR_MAX · INTMAX_MIN · INTMAX_MAX · UINTMAX_MAX

intptr_t

```
typedef signed int intptr_t;
```

Signed 16-bit integer capable of holding a converted guest pointer.

stdint.h

uintptr_t

```
typedef unsigned int uintptr_t;
```

Unsigned 16-bit integer capable of holding a converted guest pointer.

stdint.h

intmax_t

```
typedef signed long intmax_t;
```

The widest signed integer supported here, a 32-bit long.

stdint.h

uintmax_t

```
typedef unsigned long uintmax_t;
```

The widest unsigned integer supported here, a 32-bit unsigned long.

stdint.h

INT8_MIN

```
INT8_MIN = SCHAR_MIN
```

Smallest representable value of int8_t.

stdint.h

INT8_MAX

```
INT8_MAX = SCHAR_MAX
```

Largest representable value of int8_t.

stdint.h

UINT8_MAX

```
UINT8_MAX = UCHAR_MAX
```

Largest representable value of uint8_t.

stdint.h

INT16_MIN

```
INT16_MIN = INT_MIN
```

Smallest representable value of int16_t.

stdint.h

INT16_MAX

```
INT16_MAX = INT_MAX
```

Largest representable value of int16_t.

stdint.h

UINT16_MAX

```
UINT16_MAX = UINT_MAX
```

Largest representable value of uint16_t.

stdint.h

INT32_MIN

```
INT32_MIN = LONG_MIN
```

Smallest representable value of int32_t.

stdint.h

INT32_MAX

`INT32_MAX = LONG_MAX`

Largest representable value of int32_t.

stdint.h

UINT32_MAX

`UINT32_MAX = ULONG_MAX`

Largest representable value of uint32_t.

stdint.h

INTPTR_MIN

`INTPTR_MIN = INT_MIN`

Smallest representable value of intptr_t.

stdint.h

INTPTR_MAX

`INTPTR_MAX = INT_MAX`

Largest representable value of intptr_t.

stdint.h

UINTPTR_MAX

`UINTPTR_MAX = UINT_MAX`

Largest representable value of uintptr_t.

stdint.h

INTMAX_MIN

`INTMAX_MIN = LONG_MIN`

Smallest representable value of intmax_t.

stdint.h

INTMAX_MAX

`INTMAX_MAX = LONG_MAX`

Largest representable value of intmax_t.

stdint.h

UINTMAX_MAX

`UINTMAX_MAX = ULONG_MAX`

Largest representable value of uintmax_t.

stdint.h

stdio.h

Add **vcccompat** for the bounded formatters. For Disk BASIC file streams, select **vcccompat** followed by **decbfile** and include `<stdio.h>`. The guest can keep two streams open at once. Each uses the disk currently routed to its logical drive. Close all streams before changing the mounted image or its routing. Finish any BASIC file operations on that drive before opening a writer. Console printf keeps its CMOC formatting conventions.

snprintf · vsnprintf · fprintf · vfprintf · EOF · FILE · fopen · fgetc · fgets · fread · fputc · fputs · fwrite · fflush · ftell · fseek · rewind · vcc_rewind · SEEK_SET · SEEK_CUR · SEEK_END · feof · ferror · clearerr · rename · remove · fclose · vcc_file_error

FILE

```
typedef struct VCCFile FILE;
```

Each handle has its own sector buffer, position and error indicator. A reader also keeps track of end-of-file. Keep the pointer returned by fopen, then release it once with fclose.

stdio.h

fopen

```
FILE *fopen(const char *path, const char *mode);
```

Open a file on a mounted guest disk. Use an 8.3 Disk BASIC filename, optionally followed by a drive number, as in **PLANETS.TXT:0**. Drives 0 through 3 are accepted. The guest firmware determines which device answers. Use a one-to-eight-character name and, if present, a one-to-three-character extension. Spaces, wildcards and folder paths are rejected. Names are case-insensitive and default to drive 0. A null result means open failed. Check vcc_file_error for the Disk BASIC status.

Text mode	Binary mode	What happens at open
`r`	`rb`	Read an existing file from byte zero.
`r+`	`r+b` or `rb+`	Read and write an existing file from byte zero. Its contents and length are preserved. A missing file returns **DECB_ERR_NOT_FOUND**.
`w`	`wb`	Write from byte zero. Create the file if absent or immediately truncate it if present.
`w+`	`w+b` or `wb+`	Create an empty file for reading and writing. Opening an existing file immediately discards its contents.
`a`	`ab`	Keep existing contents and write at the end. Create the file if absent.
`a+`	`a+b` or `ab+`	Read existing contents from byte zero. Every write appends, even after a seek. Create the file if absent.

These are the supported modes. Text input translates CR, LF and CRLF to a C newline. Text output stores a newline as CR. Choose binary mode for fixed-size records. Before opening a write-capable handle, close any other handle to that filename. After a successful flush, a second read-only handle can open the published contents.

On an update stream, use fseek before switching from reading to writing. You can write immediately if the read reached end-of-file. To switch from writing to reading, first flush or seek successfully. A direction change without the required step fails, leaves the position intact and sets **vcc_file_error()** to **DECB_ERR_INVALID_ARGUMENT**. Mark a planet as visited or add a return trip.

stdio.h · See it used in Planet Survey

fgetc

```
int fgetc(FILE *stream);
```

Read one character at the current position and advance. Keep the result in an int until you have compared it with EOF. In binary mode a stored $FF byte yields 255, distinct from EOF. On EOF, inspect ferror to distinguish a read failure from the end of the file.

stdio.h

fgets

```
char *fgets(char *dest, int capacity, FILE *stream);
```

Supply a writable array and its capacity, including room for the final zero. A successful call returns dest and includes the newline if it fit. With capacity 1 the function returns an empty string and consumes no input. Use at least 2 to advance. A long line can take several calls. A final line without a newline can still be delivered before end-of-file. On a null result, check ferror before using the array.

stdio.h · See it used in Planet Survey

fread

```
size_t fread(void *dest, size_t size, size_t count, FILE *stream);
```

Read up to count items of size bytes into a writable buffer with room for all requested data. The return counts complete items. A short item can leave part of its data in the buffer. For a byte count, pass size 1. Choose rb when the data contains arbitrary bytes. A request for zero items or zero-sized items returns zero.

stdio.h

fputc

```
int fputc(int c, FILE *stream);
```

Write one byte to an open output stream. Returns the byte as an integer, or EOF on failure. In a text file the C newline is stored as a Disk BASIC carriage return. Use a binary mode to preserve the byte exactly.

stdio.h

fputs

```
int fputs(const char *text, FILE *stream);
```

Write a zero-terminated C string. The terminating zero stays in memory. Include a newline in the string if you want to end the disk record. A nonnegative result means the stream accepted the text. EOF signals an error.

stdio.h · Write a survey log

fprintf

```
int fprintf(FILE *stream, const char *format, ...);
```

Write formatted characters directly to a writable guest file. Use the conversions, width and precision described under snprintf. Supply a matching argument for each conversion. The return counts characters accepted by the stream. A result of −1 signals a format, range or output failure. A prefix of the record may already have been written. A C newline follows the open file's text or binary mode. Check fflush or fclose to catch errors that surface when the disk receives the buffered data.

stdio.h · Append a numbered visit

vfprintf

```
int vfprintf(FILE *stream, const char *format, va_list args);
```

Use this form inside a function that accepts a variable argument list. Pass the destination file, the format string and an initialized va_list. It uses the same conversions, return count and failure handling as fprintf.

stdio.h

fwrite

```
size_t fwrite(const void *source, size_t size, size_t count, FILE *stream);
```

Write up to **count** elements of **size** bytes from an array. Returns the number of whole elements accepted. Use size 1 for a byte count. Zero size or count returns zero. A short result can include a partially written last element. Check the result before advancing the source pointer, then check flush or close for buffered output errors.

stdio.h

fflush

```
int fflush(FILE *stream);
```

Make pending output readable on the mounted guest disk and keep the stream open so the program can write the next record. Pass NULL to flush all open output streams. A successful flush also lets an update stream switch from output to input at its current position. Passing a read-only handle or an update handle whose last operation was input returns EOF. Flushing all streams skips handles in the input phase. A zero result means success. On EOF, check ferror before closing. If a protected disk refused a flush, leave the stream open, make that same disk writable, call clearerr and retry the flush. If a long write fails, the file can contain the portion already written. The emulator's write-protection and temporary-changes settings still govern the disk image on the Mac.

stdio.h · Read the field log while its writer is open

ftell

```
long ftell(FILE *stream);
```

Get the current file position. Store it in a **long** so it still works above byte 65,535. At the beginning the position is zero. On a binary stream it counts bytes from the beginning. On a text reader, use the result as a bookmark with **SEEK_SET**. A carriage return and line feed on disk can produce just one C newline. On an output stream the position includes bytes accepted into its buffer. A return of −1 signals failure.

stdio.h · Bookmark a planet

fseek

```
int fseek(FILE *stream, long offset, int origin);
```

Move to a position in an open file. For a binary stream, choose SEEK_SET, SEEK_CUR or SEEK_END and supply a signed byte offset. Both the beginning and the exact end of the file are valid destinations. Success returns zero and clears the end-of-file indicator. An out-of-range request returns nonzero and leaves the previous position and EOF indicator intact.

For text, restore a bookmark with **SEEK_SET**, or use a zero offset to go to the start, the current position or the end. Seeking on a writer flushes pending bytes. A successful seek on an update stream permits either reading or writing next. Check the result before continuing. An ordinary writer can then overwrite part of what it has written. In append mode the next write always goes to the end. A seek past the end cannot grow the file.

stdio.h · Jump to a fixed-size record

rewind

```
rewind = vcc_rewind
```

With **stdio.h** alone, write **rewind(stream)** to return to the beginning. A source file that also includes **disk.h** can use both interfaces. In that file use **vcc_rewind** for a FILE stream and rewind for a FileDesc.

stdio.h · Planet Atlas

vcc_rewind

```
void vcc_rewind(FILE *stream);
```

Return to the start of the file and clear the stream's error and EOF indicators on success. Use this after reading the last record to read the first one again. When a program needs to check whether repositioning succeeded, use **fseek** with zero and **SEEK_SET**.

stdio.h · Start the atlas again

SEEK_SET

```
SEEK_SET = 0
```

Measure the offset from the start of the file. With fixed-size binary records, multiply the zero-based record number by the record size. Use a long operand before multiplying when the product may exceed a 16-bit integer.

stdio.h · fseek

SEEK_CUR

```
SEEK_CUR = 1
```

Measure the offset from the stream's current position. A negative offset moves a binary reader backward. Text streams accept a zero relative offset.

stdio.h · fseek

SEEK_END

```
SEEK_END = 2
```

Measure the offset from the end of the file. On a nonempty binary file, −1 selects the last byte. Zero selects the position immediately after the last byte and also works on empty files and text streams.

stdio.h · fseek

feof

```
int feof(FILE *stream);
```

Return nonzero after a read has encountered end-of-file. Reading exactly the last available byte can succeed with this indicator still clear. Drive a loop with the result of the read and inspect feof and ferror afterward.

stdio.h

ferror

```
int ferror(FILE *stream);
```

Return the saved decbfile error code when disk I/O or allocation has failed on this open stream. Inspect the indicator before closing the file. A disk that successfully delivered its last byte can reach EOF with its error indicator clear.

stdio.h · See it used in Planet Survey

clearerr

```
void clearerr(FILE *stream);
```

Clear the saved error and end-of-file indicators for this open stream. The file position, pending output and current read/write direction stay intact. After a failed flush on a protected practice disk, make the same disk writable, call **clearerr**, then check the result of a new fflush. Only a successful flush confirms the buffered bytes reached the guest disk. After clearing EOF on a reader, another read at the end can set EOF again.

stdio.h · Recover the Mars entry

rename

```
int rename(const char *oldPath, const char *newPath);
```

Give a closed guest disk file a new 8.3 name on the same logical drive. For example, **rename("ATLAS.DAT:1", "ATLAS.BAK:1")** keeps the data and changes the directory entry on drive 1. An omitted drive suffix means drive 0 on either argument. The new name must be unused. Names compare without regard to case, so a change of capitalization alone also fails. Close any streams using either name first. A successful call returns zero. On failure it returns a nonzero value and vcc_file_error gives the reason. Renaming the old atlas and installing its replacement are two separate operations. The exercise shows which name to look for if the program stops between them.

stdio.h · Recover a Planet Atlas backup

remove

```
int remove(const char *path);
```

Delete a closed file from the mounted guest disk and release its granules for reuse. An optional suffix such as **:1** selects the logical drive. There is no undo. Check that you have a complete copy under the name you intend to keep before removing a draft or backup. Zero means success. A nonzero result means failure. Check vcc_file_error to distinguish a missing file from a disk error.

stdio.h · Recover a Planet Atlas backup

fclose

```
int fclose(FILE *stream);
```

Flush pending output and release the stream slot. A successful close returns zero. Check for EOF on a failed output operation or an invalid handle. A disk can fill while flushing the last sector, even after a write appeared to succeed. The pointer is no longer usable after close. Close both handles individually if two files are open.

stdio.h · See it used in Planet Survey

vcc_file_error

```
unsigned char vcc_file_error(void);
```

Look up the most recent file failure, particularly after fopen returns a null pointer. The result is a DECB_ERR_* code from decbfile.h. A successful open clears the saved code. An update-stream direction error also appears here. For a disk I/O error on a particular open stream, use ferror.
stdio.h

snprintf

```
int snprintf(char *dest, size_t capacity, const char *format, ...);
```

Format into a caller-owned buffer. A positive capacity always leaves a terminating NUL after the stored prefix. A zero capacity writes nothing and permits a null pointer. The return is the number of bytes the complete result would occupy, excluding the terminator. A value at least as large as capacity indicates truncation. Negative means an unsupported format or a result outside the signed 16-bit range. Supported conversions are %% (a percent sign), %c, %s, %d, %i, %u, %o, %x and %X. Integer conversions accept l. Decimal width, left alignment, numeric zero padding, and string or integer precision are available. Dynamic width, floating point and %n are outside this subset.
stdio.h

vsnprintf

```
int vsnprintf(char *dest, size_t capacity, const char *format, va_list args);
```

Same bounded behavior and format subset as snprintf. Supply an initialized va_list for the arguments after the format string. Each conversion must match its promoted argument type.
stdio.h

EOF

```
EOF = (-1)
```

A negative int distinct from valid unsigned byte values. It can be passed through the character classifiers.
stdio.h

stdlib.h

Use **vcccompat** for strtol and strtoul. For allocation, also add **wydealloc** after **vcccompat** in library order. The VCC Plus allocation functions share one foreground heap inside a caller-owned guest memory region. Initialize it before requesting storage. A request can fail when free space is fragmented or the pool is exhausted. The raw WydeAlloc interface can manage additional independent pools.
vcc_heap_init · malloc · calloc · realloc · free

strtol (stdlib.h)

```
long strtol(const char *text, char **endptr, int base);
```

Read a signed 32-bit number from a zero-terminated string. Leading ASCII whitespace and a single plus or minus sign are accepted. Supply base 2 through 36 to choose the digits. Letters A through Z stand for values 10 through 35, in either case. Base zero recognizes hexadecimal **0x** or **0X** before a hex digit, octal for a leading **0**, and decimal otherwise. Base 16 also accepts the hex prefix. For binary input use base 2 and supply the digits without a prefix.

With a writable **char** pointer called **end**, pass **&end** to receive the address of the first unconverted character. If no digit was read, the result is zero and **end** points to the original input. You may pass **NULL** when you do not need the end position. On overflow the result is **LONG_MIN** (−2,147,483,648) or **LONG_MAX** (2,147,483,647) and errno becomes **ERANGE**. An unsupported base sets **EINVAL**.
stdlib.h · Unsigned conversion · CMOC decimal macro

strtoul (stdlib.h)

```
unsigned long strtoul(const char *text, char **endptr, int base);
```

Read an unsigned 32-bit number using the same digits, bases, optional sign and end-pointer rules as strtol. A leading minus sign negates an in-range magnitude in unsigned arithmetic, so **-1** yields 4,294,967,295. A magnitude larger than 4,294,967,295 yields **ULONG_MAX** and sets **ERANGE**. Clear errno before converting if you need to distinguish overflow from an exact **ULONG_MAX**.
stdlib.h · Check a planet number · CMOC decimal macro

vcc_heap_init

```
int vcc_heap_init(void *buffer, size_t capacity);
```

Give the allocator a writable region of at least 16 bytes that stays mapped at the same guest address throughout use. A static byte array works. Pass the array and its size in bytes. The address immediately after the region must also fit in 16 bits. Keep stack space and other application data outside this region. Bookkeeping takes part of the capacity. A return of 1 means success. Invalid arguments or any live allocation cause the call to return 0 and preserve the previous heap. Once every block has been released, you may initialize a new region.
stdlib.h · See the crew list

malloc

```
void *malloc(size_t size);
```

Request a contiguous block of **size** bytes from the initialized heap. Initialize its contents before reading them. A zero request, an unavailable heap or insufficient space yields a null pointer. Release each successful allocation exactly once with free unless realloc has already replaced or released it.
stdlib.h

calloc

```
void *calloc(size_t count, size_t size);
```

Request space for **count** objects of **size** bytes each. All bytes in a successful allocation start at zero. A zero argument, a product too large for a 16-bit size, or insufficient memory yields a null pointer. Pass count and size separately so the allocator can detect multiplication overflow.
stdlib.h · See the crew list

realloc

```
void *realloc(void *ptr, size_t size);
```

Resize a live allocation from this heap. Keep the returned pointer in a temporary until you check it. A nonzero request that fails leaves the original pointer and all its bytes valid. Success preserves the first bytes up to the smaller of the old and new sizes. Initialize any new bytes. Shrinking keeps the same block and reserves its underlying storage until it is freed. When growth moves the data, the old and new blocks must both fit during the copy. Passing a null pointer makes an ordinary allocation. Size zero releases the old allocation and returns a null pointer.
stdlib.h · See the crew list

free

```
void free(void *ptr);
```

Return a live allocation to the same heap. Use the pointer to the beginning of the block returned by malloc, calloc or realloc. All pointers into that block expire when it is freed. Passing a null pointer is harmless.
stdlib.h · See the crew list

string.h

Add **vcccompat** and include `<string.h>`. This header publishes the existing CMOC declarations, including strlen, strcmp, strchr, strtok, memcpy and memmove. When buffers overlap, choose memmove. Only the documented CMOC subset is available.
For a list inside a record, use strtok_r with one saved position for the record and another for the list.

strtok_r

```
char *strtok_r(char *text, const char *delimiters, char **saveptr);
```

This POSIX extension finds the next nonempty token in a writable string. Start with the string and the address of a `char *` variable. To continue, pass **NULL** for the string and the same variable's address. Each character in **delimiters** is a separator. Leading and repeated separators are skipped. An empty delimiter string takes the whole remaining text as one token. When no token remains, the call returns **NULL**.

The result points into the original buffer. The function writes a zero byte at the separator following a token, so use a writable array and finish with the returned pointers before refilling that array. Each scan gets its own saved position. In Planet Survey, this lets you resume the outer record after splitting the traits inside it.

string.h · strtok

vcc/printer.h

Select **vcccompat** under **Project Settings → Libraries**. These foreground routines call the CoCo BASIC ROM to send text to its serial printer. Each call preserves the program's regular output destination. Open **Media → Printer Output…** and choose **Start Capture** to collect the bytes on the Mac. Send the flight report.

vcc_printer_printf · vcc_printer_vprintf

vcc_printer_printf

```
int vcc_printer_printf(const char *format, ...);
```

Supply the format string and one argument for each conversion. The snprintf formatting rules apply. The printer receives a carriage return for each C newline. The result counts formatted characters handed to BASIC. A format error or a count above 32,767 returns −1. A prefix may already have gone to the printer. To check that capture was enabled, look in **Printer Output** after sending. Flight report exercise.

vcc/printer.h

vcc_printer_vprintf

```
int vcc_printer_vprintf(const char *format, va_list args);
```

Use this form inside a helper that takes a variable number of arguments. Initialize the list with **va_start**, forward it with the format string, then call **va_end** in the helper. It uses the same printer route, conversions and return values as vcc_printer_printf.

vcc/printer.h

vcc/input.h

Add **input**, then **bcontrol** to your linked libraries. Use the standard analog CoCo joystick and the usual BASIC PIA configuration. Initialize one record per port and let movement, firing and charging all read the same snapshot. Launch a survey probe.

Sampling runs in the foreground. It temporarily masks IRQ and FIRQ while it reads the axes and buttons, then restores the interrupt masks, keyboard column output, DAC value and sound routing. Reading a PIA data register can acknowledge a pending PIA interrupt. When using Orchestra-90 foreground playback, sample between blocks. Shorter blocks let the game check the controls more often.

VccInput · INPUT_RIGHT · INPUT_LEFT · INPUT_PRIMARY · INPUT_SECONDARY · input_init · input_sample

VccInput

```
typedef struct { … } VccInput;
```

Keep one of these records per player. After a sample, x and y range from 0 to 63, increasing to the right and down. The signed dx and dy fields report direction: −1 below 26, zero from 26 through 37, and +1 above 37. Use held, pressed and released as button masks. All three describe the same sample.

vcc/input.h

INPUT_RIGHT

```
INPUT_RIGHT = 0
```

Select the right CoCo joystick port when initializing a record.

vcc/input.h

INPUT_LEFT

```
INPUT_LEFT = 1
```

Select the left CoCo joystick port. Each port can have its own record and press history.

vcc/input.h

INPUT_PRIMARY

```
INPUT_PRIMARY = 1
```

Test the first fire button with a bitwise AND, for example `stick.pressed & INPUT_PRIMARY`. The expression is true on the sample where the press first appears.

vcc/input.h

INPUT_SECONDARY

```
INPUT_SECONDARY = 2
```

Test the second fire button in any of the three button fields. Both buttons can be down together.

vcc/input.h

input_init

```
void input_init(VccInput *input, unsigned char port);
```

Establish the starting position and held buttons. If a player holds fire as the program starts, held includes it and both event masks start empty. An invalid port leaves a neutral, inactive record. A null pointer is ignored.

vcc/input.h

input_sample

```
void input_sample(VccInput *input);
```

Refresh the record once per game update. pressed contains the buttons that went down since the last sample, released contains those that went up, and held contains all buttons down now. A complete tap between samples can go unobserved. A null pointer is ignored.

vcc/input.h

vcc/gime.h

Add **gime** to a CoCo 3 BASIC project to use one 320 × 192 graphics display with 16 palette slots and two pages. Start from BASIC text mode (32, 40 or 80 columns), with the usual BASIC memory map, task 0 and interrupts enabled. The library temporarily uses the upper part of the CPU address space to reach video RAM. Keep your program, data and stack below $8000. Use this API from the foreground. Finish your drawing and close the session before using BASIC services such as printing or disk I/O.

Reserve physical $40000 through $4FFFF on a machine with at least 512 KiB of RAM. Opening clears the pictures in those 64 KiB. A page has 192 rows of 160 bytes and takes up four 8 KiB memory blocks. The two pages share a hardware palette. Fly the lander to see the pages alternate.

GIME_WIDTH · GIME_HEIGHT · gime_open · gime_clear · gime_plot · gime_fill_rect · gime_set_palette · gime_present · gime_close

GIME_WIDTH

```
GIME_WIDTH = 320
```

Number of pixels in a row. The x coordinate of the rightmost pixel is GIME_WIDTH minus one.

vcc/gime.h

GIME_HEIGHT

```
GIME_HEIGHT = 192
```

Number of rows in each graphics page. The y coordinate of the bottom row is GIME_HEIGHT minus one.

vcc/gime.h

gime_open

```
unsigned char gime_open(void);
```

Begin a two-page graphics session. A result of 1 means you can draw on the back page. A result of 0 means the requested session could not start, including when a session is already active. Check the result before drawing. The physical video pages belong to this session until you close it.

vcc/gime.h · Fly the lander

gime_clear

```
void gime_clear(unsigned char color);
```

Fill the entire back page with the palette slot **color** (0 through 15). The current picture stays visible while you prepare its replacement. Clearing all 30,720 bytes every frame costs guest CPU time. For a moving object on an otherwise static picture, consider erasing just its old footprint on each page.

vcc/gime.h

gime_plot

```
void gime_plot(unsigned x, unsigned y, unsigned char color);
```

Set one back-page pixel. The origin is the upper left. Valid coordinates are x = 0 through 319 and y = 0 through 191. One byte holds two palette-slot numbers. Plotting one pixel preserves its neighbor. A BGraph drawing callback can pass its coordinates and slot directly to this function.

vcc/gime.h · Trace the lander's ridge

gime_fill_rect

```
void gime_fill_rect(unsigned x, unsigned y, unsigned w, unsigned h, unsigned char color);
```

Fill a back-page rectangle, starting at **x, y**. Its width and height are pixel counts. A width of 1 paints a single column. A height of 1 paints a single row. Drawing is clipped to the 320 × 192 picture. Choose a palette slot from 0 through 15. Zero-sized rectangles, origins outside the picture and invalid slots leave the page untouched.

vcc/gime.h · Draw the lander's cabin and landing gear

gime_set_palette

```
void gime_set_palette(unsigned char slot, unsigned char rgb);
```

Set one of the 16 hardware palette slots to a six-bit GIME color (0 through 63). The bits, from high to low, are the high red, green and blue bits followed by their three low bits. On RGB output, $00 gives black, $3F gives white and $1B gives bright cyan. A palette change immediately affects both pages, including pixels already on screen.

vcc/gime.h

gime_present

```
unsigned char gime_present(void);
```

Wait for a vertical field boundary and display the completed back page. A result of 1 means the previous front page is now yours to draw on. Its existing pixels remain there. A result of 0 means the pages did not swap. Check that a session is active and its field timing is available. In the lander, a failed presentation goes through normal cleanup and reports the problem after returning to text.

vcc/gime.h · Follow one frame

gime_close

```
void gime_close(void);
```

End the graphics session. Repeated calls after closing are safe. BASIC regains its text display and saved palette, and the CPU regains its original mapping. You can print the landing result and return from main. Data you previously kept in the two physical graphics pages is not recovered.
vcc/gime.h

vcc/bank.h

Include <vcc/bank.h> and link **bank** to copy data between a C buffer and four reserved physical RAM banks on a CoCo 3. Use the ordinary CoCo 3 BASIC memory map with at least 512 KiB. Keep the program, its stack and the buffers below $8000. Use the library from foreground code, with the standard expansion bank group selected. A program that changes the raw MMU or expansion registers must restore the BASIC setup before using this library.
Dedicate physical **$50000 through $57FFF** to the program's assets. The caller owns this reservation. An asset offset counts bytes from $50000. It can span a bank boundary in a single call. The library temporarily maps one bank into the CPU window at $8000, transfers bytes, then restores the window. Code outside the call uses ordinary pointers to its own buffers. The GIME graphics pages occupy a separate region. Run the two-landscape exercise.
BANK_PAGE_SIZE · BANK_FIRST_PAGE · BANK_CAPACITY · bank_open · bank_read · bank_write · bank_close

BANK_PAGE_SIZE

```
BANK_PAGE_SIZE = 8192u
```

Number of bytes in one MMU bank. In the terrain exercise this is also the offset of the first Mars byte. The last Earth-bank byte is at offset 8,191.
vcc/bank.h

BANK_FIRST_PAGE

```
BANK_FIRST_PAGE = 0x28
```

Physical number of the first reserved bank. Multiply by $2000 bytes per bank to find its starting address in the debugger. Use offset **zero** to reach this location in a transfer.
vcc/bank.h

BANK_CAPACITY

```
BANK_CAPACITY = 32768u
```

Capacity of all four banks together, in bytes. The last asset byte has offset 32,767. A two-byte transfer from there fails before either byte is copied.
vcc/bank.h

bank_open

```
unsigned char bank_open(void);
```

Start a bank session before the first transfer. Check for 1 before using it. Call with IRQ enabled. A repeated open or a failed mapping or memory check returns 0. On expanded machines the bank selector at $FF9B is write-only, so the caller must retain its normal BASIC setting. Opening preserves existing contents, so initialize the bytes your program plans to read.
vcc/bank.h

bank_read

```
unsigned char bank_read(unsigned offset, void *destination, unsigned length);
```

Copy **length** bytes from the reserved banks into your writable C buffer. Check for 1 before using the result. A refused request returns 0 and leaves the destination alone. Supply a buffer large enough for the whole transfer. The library can check the address range but cannot determine a C array's declared length.
vcc/bank.h · Fetch a landscape

bank_write

```
unsigned char bank_write(unsigned offset, const void *source, unsigned length);
```

Copy **length** bytes from your C buffer into the reservation, starting at **offset**. Returns 1 on success and 0 on refusal. A transfer may continue into the next reserved bank. A request extending past the reservation changes no asset bytes. In an open session a zero-length transfer at the end of the reservation is valid.

vcc/bank.h · Raise the Mars ridge

bank_close

```
void bank_close(void);
```

End the session. Closing preserves the bytes in RAM and is safe to repeat. A later program or a power cycle can change that memory. To keep an edit across sessions, save the data to guest media.

vcc/bank.h

vcc/sdc.h

Include `<vcc/sdc.h>` and link **sdc**. Use foreground calls from a low-memory CMOC program with the normal CoCo 3 BASIC map. The supported configuration uses the bundled 16 KiB SDC-DOS ROM and routes cartridge ROM and I/O to the same Multi-Pak slot, as in the CoCo SDC profile. The library leaves the Multi-Pak selection and the current card directory alone.

Browse a card with **sdc_list_begin** and **sdc_list_next**. Inspect the selected disk with **sdc_mounted**. Close guest files before changing a disk with **sdc_mount**. Open a file inside the selected image with fopen. Fly Planet Expedition.

SdcEntry · SDC_DIRECTORY · SDC_READ_ONLY · SDC_OK · SDC_END · SDC_UNAVAILABLE · SDC_TIMEOUT · SDC_INVALID · SDC_IO · SDC_NOT_FOUND · SDC_IN_USE · sdc_probe · sdc_list_begin · sdc_list_next · sdc_mounted · sdc_mount · SDC_PROTECTED

SdcEntry

```
typedef struct { … } SdcEntry;
```

A caller-owned result. **name** holds a terminated 8.3 basename, including the dot when there is an extension. **attributes** is a bit mask. **size** is a 32-bit byte count. Read the fields after a successful call. Save a copy if you reuse the same variable for your next result.

vcc/sdc.h

SDC_DIRECTORY

```
SDC_DIRECTORY = 0x10
```

Test this bit in **attributes** before offering to mount an entry. A directory can be browsed by passing its path and a pattern to sdc_list_begin.

vcc/sdc.h

SDC_READ_ONLY

```
SDC_READ_ONLY = 0x01
```

The entry carries the read-only attribute. A mounted image can still be read from a protected card. Check each disk operation for its own result.

vcc/sdc.h

SDC_OK

```
SDC_OK = 0
```

The operation succeeded. During enumeration this means **entry** describes one item, so the program can use it and request the next.

vcc/sdc.h

SDC_END

`SDC_END = 1`

The enumeration has reached its end. A pattern with no matches is a normal empty list. Start a fresh enumeration to browse again.

vcc/sdc.h

SDC_UNAVAILABLE

`SDC_UNAVAILABLE = 2`

The library could not identify a supported SDC-DOS setup on the active cartridge routes. In VCC Plus, select the CoCo SDC profile before running the program.

vcc/sdc.h

SDC_TIMEOUT

`SDC_TIMEOUT = 3`

The controller did not reach the expected state within the polling limit. Stop issuing dependent commands and query the drive again before opening a file.

vcc/sdc.h

SDC_INVALID

`SDC_INVALID = 0x84`

Check the drive number, argument, path and candidate image. A rejected image leaves the previous disk available in VCC Plus.

vcc/sdc.h

SDC_IO

`SDC_IO = 0x88`

The controller reported an input/output failure. Check the selected card folder and media before retrying.

vcc/sdc.h

SDC_NOT_FOUND

`SDC_NOT_FOUND = 0x90`

The requested file or folder could not be found. After a missing-image error from sdc_mount, the program can query the drive and read the disk that was already there.

vcc/sdc.h

SDC_IN_USE

`SDC_IN_USE = 0xa0`

The requested change conflicts with a disk already in use. Close files on the target drive and check whether the other drive already has this image mounted.

vcc/sdc.h

sdc_probe

```
unsigned char sdc_probe(void);
```

Call this before presenting card controls. Success means the guest can reach a supported SDC-DOS controller. A subsequent directory or image operation can still fail if the card folder is unavailable.

vcc/sdc.h

sdc_list_begin

```
unsigned char sdc_list_begin(const char *pattern);
```

Begin a card-directory listing. An absolute pattern such as **/*.DSK **matches disk images at the root. Use** / TRIPS/** followed by a pattern to inspect a subdirectory. A null or empty pattern lists the current directory. Only

one listing is active at a time. Card paths use printable ASCII and fit in 253 bytes. Starting a second listing replaces the first. Finish the listing before mounting an image, because the mount uses the controller's directory cursor.

vcc/sdc.h

sdc_list_next

```
unsigned char sdc_list_next(SdcEntry *entry);
```

Supply a writable entry. While the call returns SDC_OK, use the name, attributes and byte count. SDC_END finishes the loop. Treat other statuses as errors. The library fetches another controller block when a directory has more than 16 entries.

vcc/sdc.h

sdc_mounted

```
unsigned char sdc_mounted(unsigned char drive, SdcEntry *entry);
```

Supply drive **0** or **1** and an entry for the result. On success **name** holds the image's basename and **size** counts its bytes on the card. The controller supplies the basename even for an image in a subdirectory. Keep the original card path separately if you will need to remount it. SDC_NOT_FOUND means the drive is empty.

vcc/sdc.h

sdc_mount

```
unsigned char sdc_mount(unsigned char drive, const char *path);
```

Mount an existing image on drive **0** or **1**. Use an absolute card path such as **/EARTH.DSK** for a predictable location. First close every guest file on that drive. After success, open files on the replacement disk. In VCC Plus, a missing, invalid or already-mounted candidate leaves the previous image in place. Try a missing planet. Finish any decbfile session that caches the old disk's allocation table. The library checks for a regular file before asking SDC-DOS to mount it. This prevents a folder named **ALBUM.DSK** from being treated as a set of disks.

vcc/sdc.h

SDC_PROTECTED

```
SDC_PROTECTED = 0xc0
```

The controller reports write protection. The library can list and mount readable images on a protected card. Check the mounted entry's SDC_READ_ONLY bit when deciding whether to offer editing.

vcc/sdc.h

vcc/cassette.h

Select **vcccompat** in **Project Settings → Libraries**. One foreground program can have one cassette file open. The tape moves forward as you read. Set up the transport in **Media → Cassette…** before each trip. Flight Recorder shows how to write a file in C and read it from BASIC, then switch roles.

vcc_tape_open_write · vcc_tape_write_line · vcc_tape_open_read · vcc_tape_read_line · vcc_tape_close
VCC_TAPE_OK · VCC_TAPE_EOF · VCC_TAPE_IO · VCC_TAPE_INVALID · VCC_TAPE_LONG · VCC_TAPE_BUSY · VCC_TAPE_TYPE

vcc_tape_open_write

```
int vcc_tape_open_write(const char *name);
```

Choose **New Recording → CAS** before calling. Supply a guest filename of one to eight printable ASCII characters, such as **FLIGHT**. Use the same letter case when you open the file for reading. BASIC stores the spelling inside its filename string. The call writes its name block to the cassette. Check for **VCC_TAPE_OK** before writing lines, then close the file after the last line. Use **Export Tape…** to keep the recording on your Mac.

vcc/cassette.h

vcc_tape_write_line

```
int vcc_tape_write_line(const char *line);
```

Send one observation as a zero-terminated string of printable ASCII characters. The routine adds a carriage return so BASIC can use **LINE INPUT #−1** to read that observation. Pass the text without a line ending. An empty string records a blank line. A file with no calls to this routine contains zero lines.

vcc/cassette.h

vcc_tape_open_read

```
int vcc_tape_open_read(const char *name);
```

Choose **Play** and supply the name the writer recorded. The tape search starts at the current position. Rewind before looking for a name you have already passed. Start reading once this call returns **VCC_TAPE_OK**.

vcc/cassette.h

vcc_tape_read_line

```
int vcc_tape_read_line(char *line, unsigned capacity);
```

Read the next complete line into a writable array. Pass the array size, including space for a trailing zero. A return of **VCC_TAPE_OK** means the array holds the record without its carriage return. A blank line returns that success code with an empty string. If BASIC suppressed the last carriage return with a trailing semicolon on **PRINT**, a verified end block completes that final string. The next read returns **VCC_TAPE_EOF**. **VCC_TAPE_EOF** means the tape contained its recorded end block. A negative return means reading failed. If a line was too long, the routine consumes that line and clears the array. You can read the next line. Check the result before using the array.

vcc/cassette.h

vcc_tape_close

```
int vcc_tape_close(void);
```

Finish the open file. On a writer, this sends any buffered data and the end block. A successful return means the CoCo finished sending them. Check the Cassette window and export the recording to keep it on the Mac. On a reader, close after reaching the end or after an error. To start reading the file again, rewind in the Cassette window and choose **Play**.

vcc/cassette.h

VCC_TAPE_OK

```
VCC_TAPE_OK = 0
```

The operation succeeded. From **vcc_tape_read_line**, this means one record was read. The record may have zero characters.

vcc/cassette.h

VCC_TAPE_EOF

```
VCC_TAPE_EOF = 1
```

The reader verified the file's end block. Use this result to end a read loop normally.

vcc/cassette.h

VCC_TAPE_IO

```
VCC_TAPE_IO = (-1)
```

Reading failed. A missing end block, an incomplete data block or an invalid checksum can produce this result. Close the reader and try again with a complete recording.

vcc/cassette.h

VCC_TAPE_INVALID

```
VCC_TAPE_INVALID = (-2)
```

The arguments or operation were invalid. Check the filename, text, or buffer capacity and confirm that the file is open in the intended direction.
vcc/cassette.h

VCC_TAPE_LONG

`VCC_TAPE_LONG = (-3)`
The line exceeds the destination capacity, including its trailing zero. The reader consumes the oversized line and returns an empty string. Supply a larger buffer for longer observations.
vcc/cassette.h

VCC_TAPE_BUSY

`VCC_TAPE_BUSY = (-4)`
A C or BASIC cassette file is already open, or the tape motor is active. Finish the operation and close any open file before trying again.
vcc/cassette.h

VCC_TAPE_TYPE

`VCC_TAPE_TYPE = (-5)`
The name matched a file of an unsupported type. The reader expects a sequential ASCII data file recorded with gaps between its blocks.
vcc/cassette.h

vcc/decimal.h

Select **vcccompat** under **Project Settings** → **Libraries** and use the default CoCo BASIC floating-point backend. The destination belongs to your program. You can print a formatted field with the ordinary **%s** conversion. Descent Computer uses the same speed once for display and once to estimate a landing.
vcc_fixed

vcc_fixed

```
int vcc_fixed(char *dest, unsigned capacity, float value, unsigned char places);
```
Write a signed decimal number with zero to three places after the decimal point. The input magnitude must be less than 65,536. A tie rounds away from zero, so −1.875 at two places becomes **−1.88**. A value that rounds to zero has no minus sign. The rounded output may reach 65,536 in magnitude. CMOC's BASIC float stores a binary value. Rounding acts on that stored number, which can differ slightly from the decimal a person typed.
The result is the full character count, excluding the terminating zero byte. Positive capacity reserves one byte for that terminator. A short buffer holds the prefix that fits. To measure first, pass a null destination with zero capacity. An invalid destination, precision or input range returns −1 and empties a supplied buffer if its capacity is positive. A complete field requires a nonnegative result strictly smaller than the capacity.
vcc/decimal.h · See it used in Descent Computer

CMOC at the desk

Appendix B

Keep this page open when a familiar C expression behaves differently on the CoCo, or when the compiler succeeds and the guest still does something surprising.

Choose a size before a value

CMOC type on this target	Bytes	Use in this book
unsigned char, also called byte in coco.h	1	A crate count or a status from the disk library, 0 through 255
signed char	1	A small quantity that may be negative, −128 through 127
int or short	2	Signed arithmetic from −32768 through 32767
unsigned int, also called word in coco.h	2	Cargo totals, 0 through 65535
long or unsigned long	4	Larger counters; signed and unsigned ranges differ
Pointer to guest memory	2	The address of a manifest row in the CoCo's current 64 KB address space

A numeric **2**, the character **'2'** and the two-byte string **"2"** have different jobs. The input loop compares a single character. The string contains the character followed by a zero byte. Printing a string asks the library to keep reading until that zero; passing it a single character value would supply the wrong kind of argument.

Choosing a wider destination cannot recover bits already lost in an earlier calculation. The cargo source explicitly casts the first factor to word before multiplication. Read the input types, the type of the calculation and the destination in that order.

Decide who owns the storage

Think about three addresses in Cargo Count. The manifest has program lifetime. A row pointer handed to mealsFor stays valid because that array continues to exist while the helper runs. The parameters and ordinary local variables of the helper belong to that invocation; their storage can be reused after it returns.

An array expression generally supplies a pointer to its first element when passed to a function. In main, sizeof can measure the actual manifest array. In a function receiving only a pointer, sizeof that pointer measures the address itself. Pass a count or capacity separately when the receiving routine needs one. Index zero is the first element; the element count is already one past the last valid index.

CMOC compiles for a much smaller stack than a Mac. Large temporary arrays and deep recursion compete for space. Look at the static 256-byte buffer in the disk reader for one way to reserve a buffer for the life of a program. Static storage is shared across calls; it would be the wrong place for two independent readers that must each keep their own in-progress data.

For storage whose size you choose while a program is running, reserve a static byte array and hand it to vcc_heap_init. Size the array alongside the executable, other globals and stack in the guest's 64 KiB address space. Adding more physical RAM to the emulated machine doesn't enlarge a C pointer. Grow a crew list to see how a program keeps its data when it runs out of space.

Know the language this compiler accepts

CMOC compiles a C-like language for the 6809. Its compiler implementation runs on the Mac. The program you write runs on the emulated processor. The same 6809 code can also run on the HD6309.

- Define structures and typedefs at file scope. Define the structure before accessing its members; a pointer can be declared earlier using an incomplete structure type.
- CMOC supports 32-bit long integers. Code assuming a 64-bit long or a long long needs adapting.
- On this CoCo target, the normal float backend uses routines in BASIC and its five-byte representation. Double is an alias for float in CMOC. Spelling double does not increase precision. An alternative MC6839 math backend needs a separately staged compatible library and its own format; check your toolchain edition before choosing it.
- A POSIX process, Unix path and host stdio stream imply an operating system. Planet Survey opens a stream on a named guest drive. Hello Disk uses decbfile to read the same disk format by sector. Look up the callable functions in Find a call.

If you want your program to run as an OS-9 command with standard output and a process exit status, follow Compile a small C program in NitrOS-9. It takes you from a C file on HD1 through the guest compiler, assembler and linker to a runnable module.

Set up a debuggable build

Use **Generate from settings** for a personal project whose rules the app maintains. **Sources** identifies C units and include directories. **Libraries** supplies the implementation archives and their order. **Resources** produces data. **Disk & Run** selects which files to put on a floppy and how to launch. **Preview** shows the Makefile before replacement. Keep a hand-authored Makefile in **Custom Makefile** mode.

The Hello Disk Makefile uses an origin of hexadecimal 2800 and a stack allowance of 2048 bytes. Debug selects **-O0** and **-fno-omit-frame-pointer**. The intermediate listing and link map relate guest instructions back to source lines; a preserved frame gives the debugger a base for locals and callers. In Release, optimization can move or eliminate work. A variable that is unavailable in a stopped frame has no dependable value to display there.

In **C Step**, **Into** enters a mapped call, **Over** lets it finish and **Out** follows the return to the caller. Select a caller in **Call Stack** to see its source and frame-specific variables. To move execution, use a step or Continue. Selecting a different frame only changes where you look.

Keep custom-Make debug evidence

A successful custom recipe proves that its command completed and its declared products passed validation. C source debugging additionally needs evidence tying those exact executable bytes to your sources. Keep matching linked `.o` files, the linker `.map`, assembler `.lst`, and CMOC intermediate `.s` files. Use `--intermediate` when compiling and linking; a Debug compile normally uses `-O0 -fno-omit-frame-pointer`. Preserve those outputs with the build they describe. A map from a different build cannot establish a source stop.

For a program that packs, relocates or banks code after linking, its project must also declare matching runtime images and physical bank identities. The app does not treat a failed bank verification as an ordinary flat program. A verified function entry can be useful even when no C statement is mapped inside the function, but it does not establish a C stack frame or dependable locals. Check which kind of stop is available before planning a variable inspection.

When an assistant uses the C connection, its build receipt distinguishes verified source debugging, source-only evidence and unavailable evidence. It can inspect structured diagnostics and retrieve the retained build log in pages. A zero Make exit status can still accompany a later product-validation failure. Ask for the failing stage and first actual error rather than treating a clipped preview or an early warning as the whole result. The native **Issues** and **Build History** controls remain useful for your own review.

Refresh a project changed outside the editor

The project input inventory distinguishes source, headers, resources and approved support folders from generated products. A directory named `generated` can contain real source; do not assume its contents are

disposable. Explicit resource/generated inputs and output directories should describe the project as its Makefile actually uses it. Merely touching an unchanged file does not create a different content revision.

An assistant's refresh request adopts clean files already covered by the connection. New files or changed recipes and support scope require the native review window. Check the paths and before/after changes there. Approval updates the authorized input set without executing it. If an external edit conflicts with an unsaved editor draft, preserve the draft and resolve the conflict locally; refresh does not silently discard or combine it. Work with an assistant explains connections and disposable tests.

Find a name and complete a call

Use **Find in File** (⌘F) for text in the selected document. **Find in Project** (⇧⌘F) searches across the project. When you want the declaration behind a name, put the caret on the identifier and choose **Go to Definition**. If the navigator offers several definitions, compare the filename, line and any **Conditional candidate** label. A custom Makefile or preprocessor condition can leave more than one plausible answer. Use Back to return to the place you started.

Choose **Complete** while typing an identifier. Use the arrow keys and Return to choose a name, Escape to dismiss it, or Undo to reverse the insertion. Once you type the opening parenthesis of a recognized call, the parameter hint shows the current argument. The hint helps you read the call while the compiler remains the check on the finished program. The editor's **Edit** menu also offers indentation, comment toggling, and navigation to a matching bracket. These act on source text, even when the CoCo is paused elsewhere.

Recover source or an earlier build

When you return to an interrupted session, choose **Recovery** in C Development. Each entry gives the date and affected filenames. Choose **Restore** to bring the draft back into the editor, inspect it, then use **Save All** when you want to update the project. **Export…** keeps a separate copy, including when the original project has moved. A record marked **Open in another app session** can be exported while its owner still has it open.

For a failed or cancelled compilation, open **Build History** instead. The **Source**, **Build** and **CoCo** status fields describe three different things. Cancelling a build leaves the program already running in the CoCo in place. An older successful build can remain in history even though the edited project now needs rebuilding. **Move Selected to Trash…** applies to eligible build results. The workspace protects the latest runnable result, the launched result, and results with assigned media.

Decide what to repair

Symptom	First place to look
Include file missing	Compare the exact filename and case with the installed header, then check the include directory.
Symbol unresolved at link	Check spelling, target, library choice and archive order. A declaration alone supplies no machine code.
No rule for a target	Check the existing Makefile. Leaving a custom project's build target blank uses Make's default goal.
Executable starts, words missing	Read the disk result and use DIR on the mounted floppy. Compare the guest filename with the one passed to the open call.
Breakpoint doesn't bind after editing	Save the sources, use a Debug build and launch the corresponding artifact. Set the stop on an executable statement.

Symptom	First place to look
Source changed outside the editor	Preserve an unsaved draft with **Save Selected Copy…** before choosing whether to reload the on-disk version. Use **Build History** to identify the earlier artifact.

For the Mac controls and their recovery options, see the C workbench in the User's Handbook.

Source shelf and further reading

Appendix C

Take the examples with you. Every program here can be studied offline, including its original source.

Read the original tool manuals

The library includes Pierre Sarrazin's complete CMOC 0.1.101 manual, the complete LWTools 4.25 manual, and the bundled tools' command-line help. Use these for exact compiler, assembler and linker options; this book remains the guide to VCC Plus's C workspace and APIs. The references identify the tool versions, preserve the authors' notices and provide original source downloads. They are searchable through **Ask the Manuals** and a project's MCP manual tools.

Get a working copy

- Earth or Mars is the single-file **Hello World** starter. Cargo Count is its companion experiment. For either, create a new **Hello World** project, replace the text in **main.c** with the example, save and build. Keep the filename expected by the Makefile.
- Save the complete Hello Disk project. Expand the ZIP in Finder and choose **Open** in C Development to select the **HelloDisk** directory that contains the Makefile. This is the same source supplied by **New → Hello Disk**. Its word list, header and generator travel with it. Build the disk before choosing Launch Disk.
- Save Planet Survey. Read a planet record across two sectors, then split its name, traits and orbit number. The project includes the editable text file and the recipe that packs it onto the disk.
- Read yesterday's log and add a return trip. Save returntrip.c to record Earth and Mars on successive launches and retry a protected save.
- Send a flight report to the TP-10. Save flightreport.c to send Earth and Mars observations to Printer Output while keeping the crew's prompt on the CoCo display.
- Exchange a flight recording with BASIC. Save Flight Recorder in C, then try its tape with the BASIC reader. The BASIC writer supplies fresh observations for the return flight.
- Keep a field log as you fly. Save fieldlog.c to write two observations and read the first one back before ending the flight.
- Save Planet Expedition. Build a practice SDC card with a program disk and two landing reports, then swap planets from C.
- Rescue an interrupted atlas save. Save atlasbackup.c and try restarting between the two disk renames on a practice copy.
- Planet Atlas. Save atlas.c to read a destination in decimal or hexadecimal, jump to its disk record and mark it so the next trip remembers your visit.
- Give Zora a voice. Save announcer.c to store a custom pronunciation, repeat a crew message and trigger a fading tone from a joystick.
- Read the Descent Computer. Save descent.c to turn two radar readings into a signed descent speed and a time to touchdown.
- Steer the Stereo Flyby. Save flyby.c to fly a ship while Orchestra-90 repeats a two-note stereo beacon.
- Fly the page-flipping lander with the right joystick or WASD. Save lander.c for the graphics exercise.
- Launch a survey probe with a joystick button. Save probe.c to try the press, charge and release exercise.
- Give the probe a second landscape. Save ridge.c to switch between two terrain maps held in banked RAM.
- Save Talking Planet Lander. Open the expanded project for the two-cartridge landing exercise. The Makefile and sources travel together.
- Save Mission Recorder. Open the expanded project to mark a sector, save its flight report and send a verified copy to the printer.
- Choose a library for a job has the individual experiments and their library settings. Each source link saves a personal copy. Opening a link in the native reader leaves the decision to build or run with you.

When reading the standalone HTML in a browser, the same files sit beside this page in the **examples** folder. If your browser displays C as text, use its download command or copy the file from that folder in Finder.
If you came from BASIC, compare the single-key C starter with the BASIC planet menu. To carry a finished image to another machine, use the six-word handoff.

Bibliography

- Sarrazin, Pierre. *The CMOC C-like 6809-targeting cross-compiler*. Version 0.1.101, November 16, 2026. See "C Language Features," "The standard library," "Floating-point arithmetic," "Detecting stack overflows at run time," and "Calling convention." Use the manual with the headers from the same edition.
- Sarrazin, Pierre. *decbfile*, 0.1.13. Header and sector-file examples. The caller-owned drive table, file handle and logical sector numbers explain the three different things passed to the open and read calls.
- Sarrazin, Pierre. *BGraph*, 0.1.9; *BControl*, 0.1.6; *BSound*, 0.1.4. Libraries and accompanying header documentation. See the drawer, joystick and player interfaces used in the optional experiments.
- Sarrazin, Pierre. *WydeAlloc*, 0.1.0; *HiresTxt*, 0.5.1; *AStar*, 0.1.0; *CardGame*, 0.1.15; *Ebitor*, 0.1.1. Source distributions and public headers. See each distribution's acknowledgments and the installed notices for additional contributors.
- Sarrazin, Pierre. *autostart.c*. Version 0.1.2, 2018. See also Schrag, Roger. "A Special Use For The DOS Command." *The Rainbow*, November 1984, p. 140. The generated Hello Disk uses an adapted startup command with the loader.
- Lindner, tim. *CoCo SDC User Guide*. Fourth version, February 2023, ch. 7, pp. 28–33. Hardware and software by Darren Atkinson. See the card directory and image commands.
- Spectral Associates. *Disk BASIC Unravelled II*. Revised by Walter K. Zydhek, December 26, 1999. See appendix B, pp. B10–B11, and program listing LC71E and LC79D for the Disk BASIC routines that record and reload the file allocation table.
- *Color Computer Disk BASIC Quick Reference Guide*. InterTAN Canada Ltd., 1987, pp. 3–9. Compare the directory, binary load and disk error entries with the commands the friend uses at the end of the book.
- Tandy Corporation. *Orchestra-90 CC Stereo Music Synthesizer with Percussion for Tandy Color Computer*. Special Composer's Edition, 1983, catalog 26-3143, pp. 66–68 and appendix E, p. 103. See the original stereo instrument used here as a pair of digital-to-analog converters.
- Tandy Corporation. *Color Computer Speech/Sound Cartridge Owner's Manual*. 1984, pp. 12–14, 17–23, 26–29, 35–36. The stored speech and sound commands and allophone chart underpin Mission Announcer.
- Tandy Corporation. *Color Computer 3 Service Manual*, section V, figure 5-7, "MMU Block Diagram." See how a CPU address selects a block of physical RAM.
- Spectral Associates. *Color BASIC Unravelled II*. Revised by Walter K. Zydhek, 1999, "Console Input/Output," pp. 13–14. The cassette buffer explains how a stream of characters travels between BASIC and tape in blocks.
- Tandy Corporation. *Tandy Color Computer 3 BASIC Quick Reference Guide*. 1986, commands, pp. 2–17. For the BASIC counterparts of the CoCo 3 screen routines.

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

Credits and toolchain editions

The VCC Plus manual prose and original instructional images are © 2026 Brian K. Holman. The examples retain their source notices. CMOC, its runtime, the companion libraries, the disk driver and the guest firmware have separate histories and terms. **Licenses & Notices** in VCC Plus identifies the components supplied in your edition.

Several of the companion libraries and the selected ROM-backed decbfile core carry public-domain dedications. WydeAlloc and Ebitor carry GPL-3.0-or-later. The original VCC Plus SDC, banked-memory, joystick input, GIME graphics, Speech/Sound, Orchestra-90 and C compatibility libraries, and the Talking Planet Lander, Planet Survey, Planet Expedition and Mission Recorder examples, also carry GPL-3.0-or-later. CMOC's compiler

and individual runtime files have their own notices. The ROM-backed disk library calls the resident Disk BASIC firmware. The optional standalone disk driver includes separately identified Disk BASIC-derived material. Neither path transfers rights in that firmware. 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.

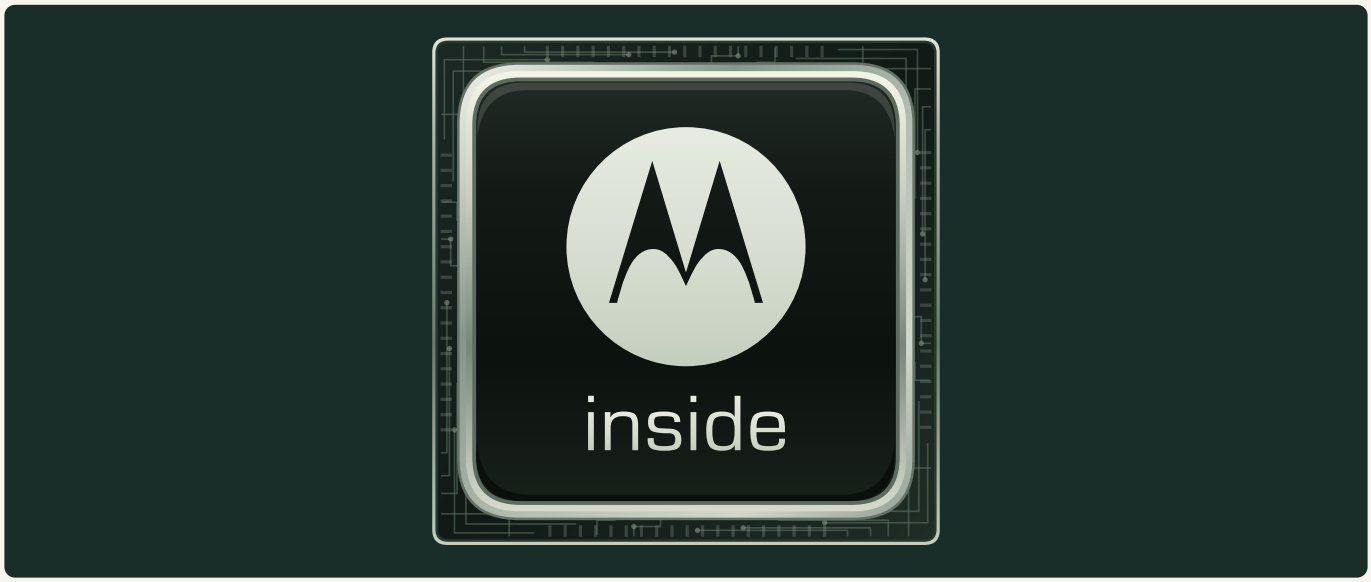
The CoCo 3 and CM-8 monitor presentation in VCC Plus's skeuomorphic System view was inspired by SockMaster's "pic of CoCo 3 and CM-8 monitor," published on CoCopedia. VCC Plus's artwork was designed and rendered independently. The remaining hardware illustrations were independently designed using historical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

Read this book online

CMOC license notice · LWTools COPYING notice

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.

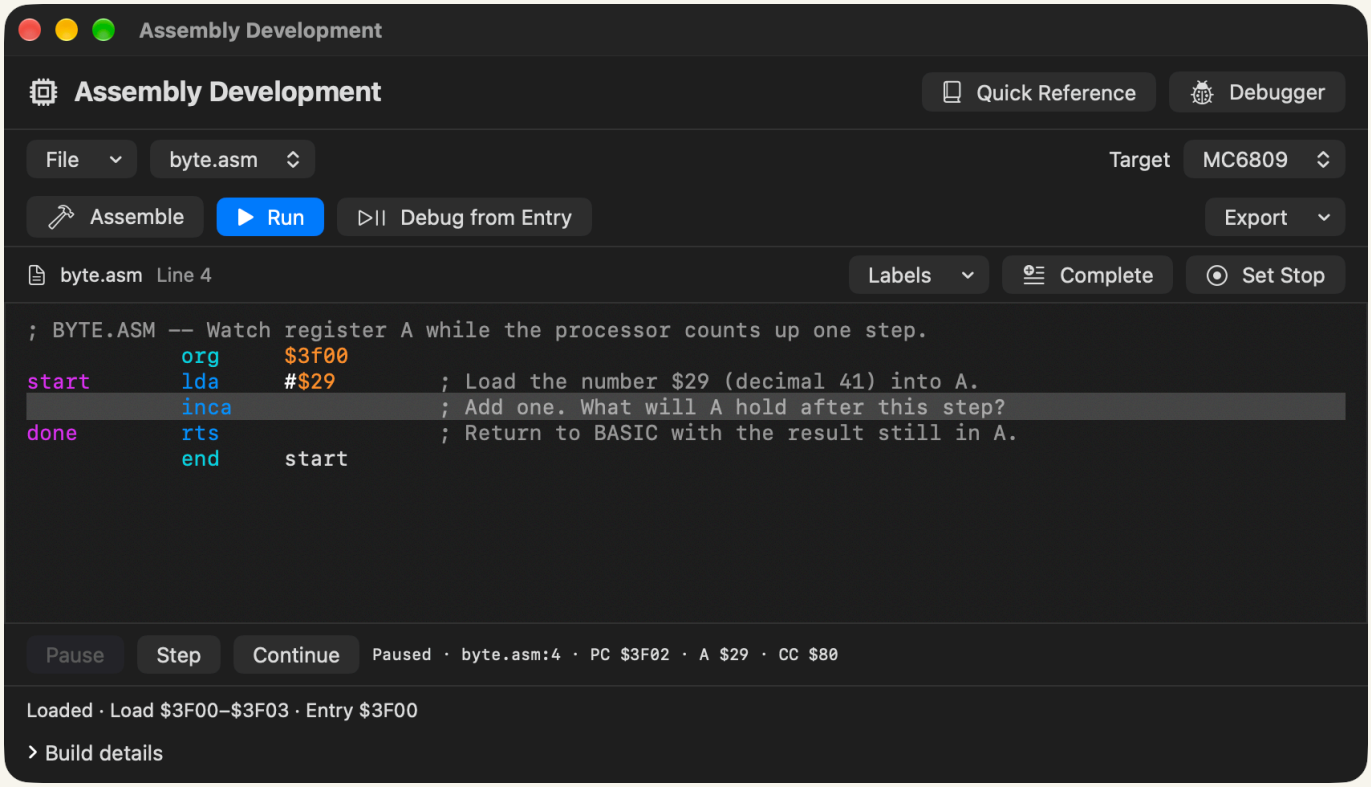


Figure 1: The assembly workspace pauses at INCA. The footer shows PC $3F02 and A $29 after executing the load.

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.

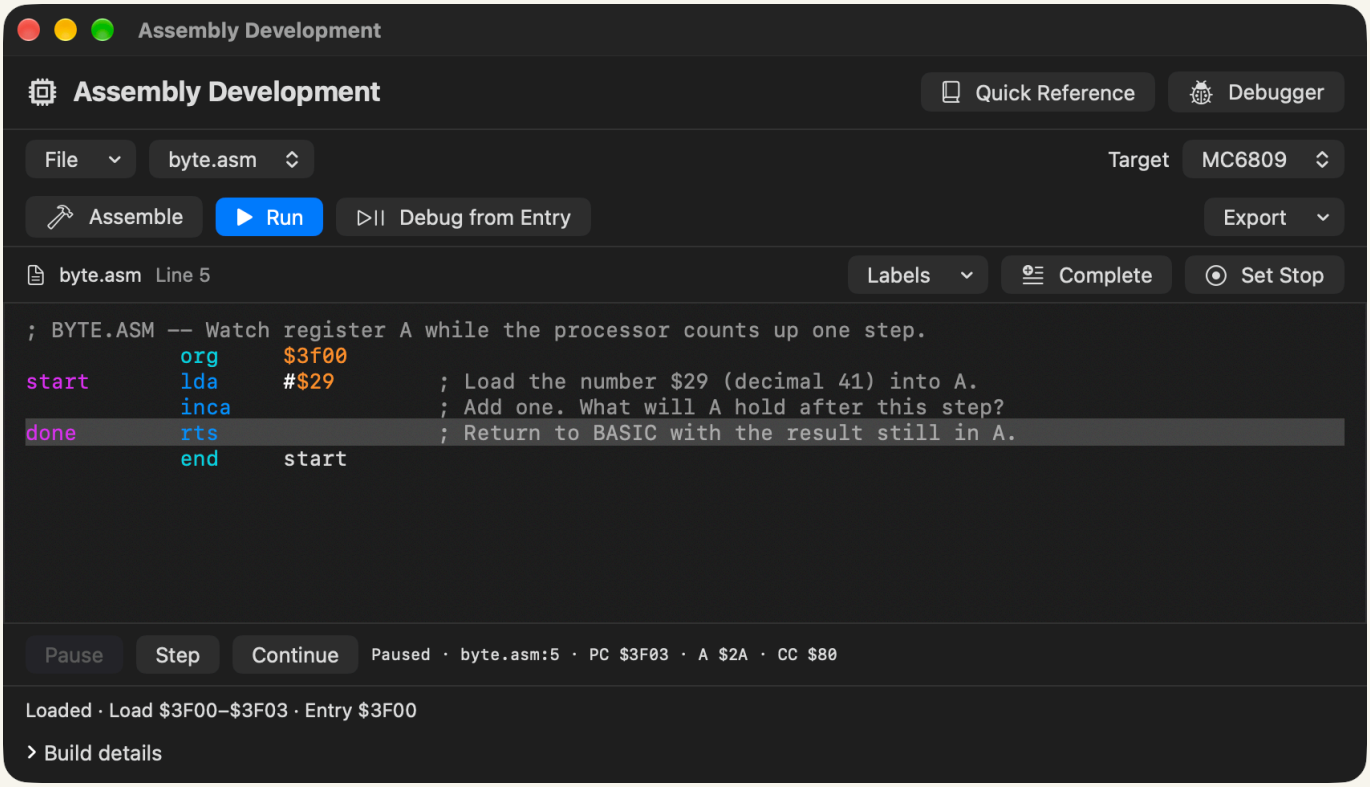


Figure 2: The next step has reached RTS. The footer shows PC $3F03 and A $2A, the result of INCA.

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.

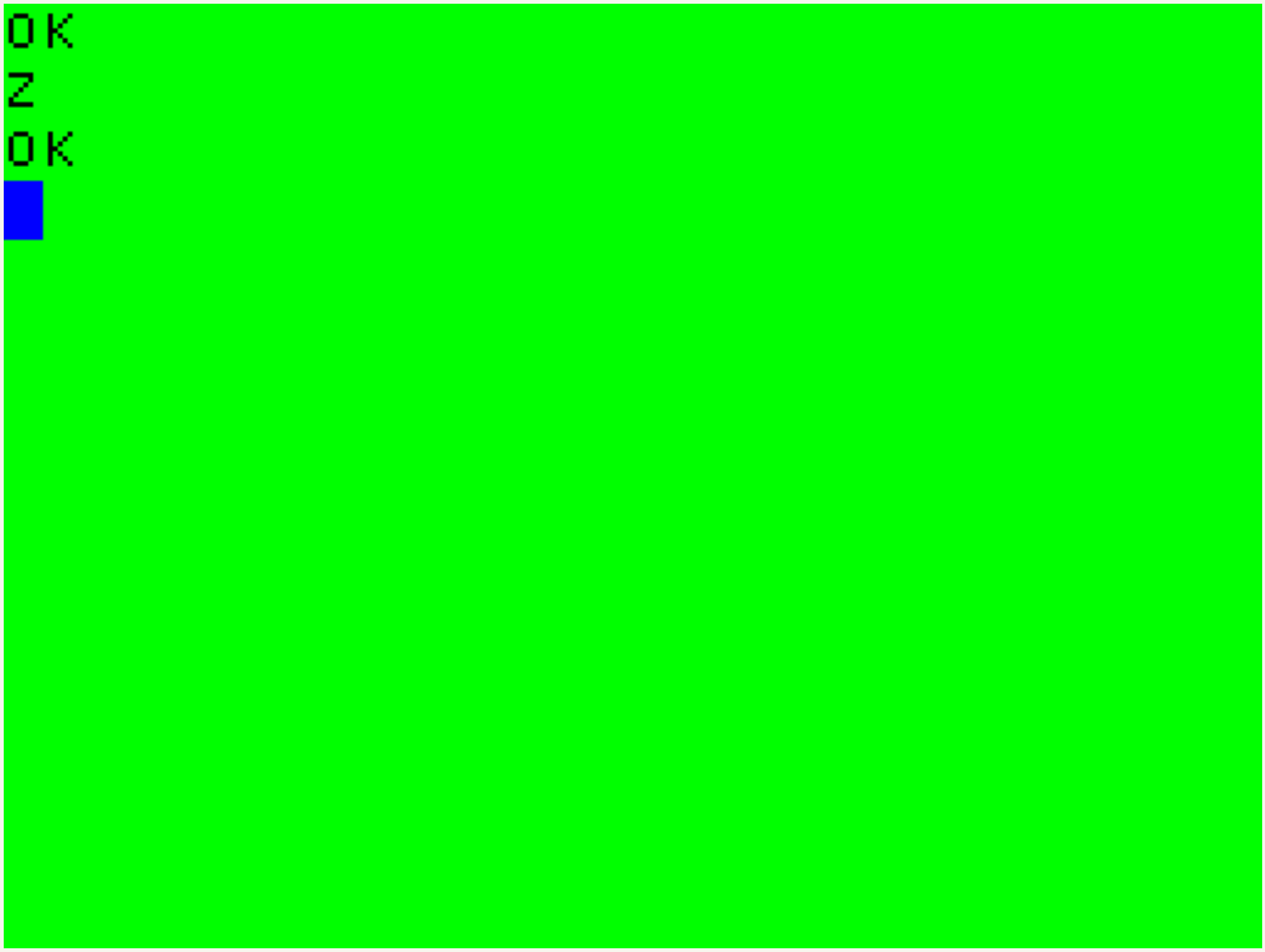


Figure 3: The CoCo screen shows Z on its own line, followed by the BASIC prompt, after running the edited character program.

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**.

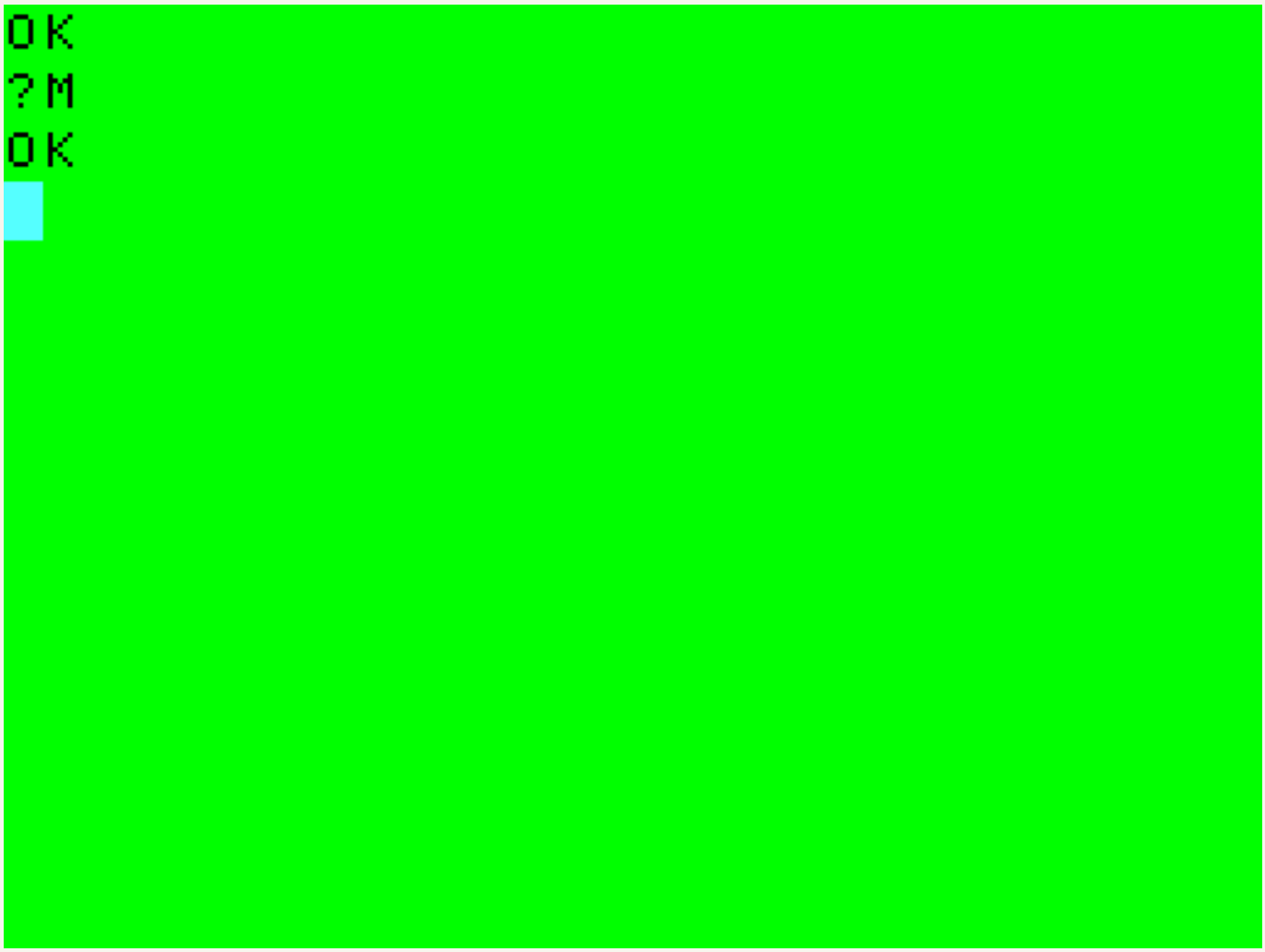


Figure 4: The CoCo display shows ?M and returns to BASIC after the Mars key was accepted.

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.

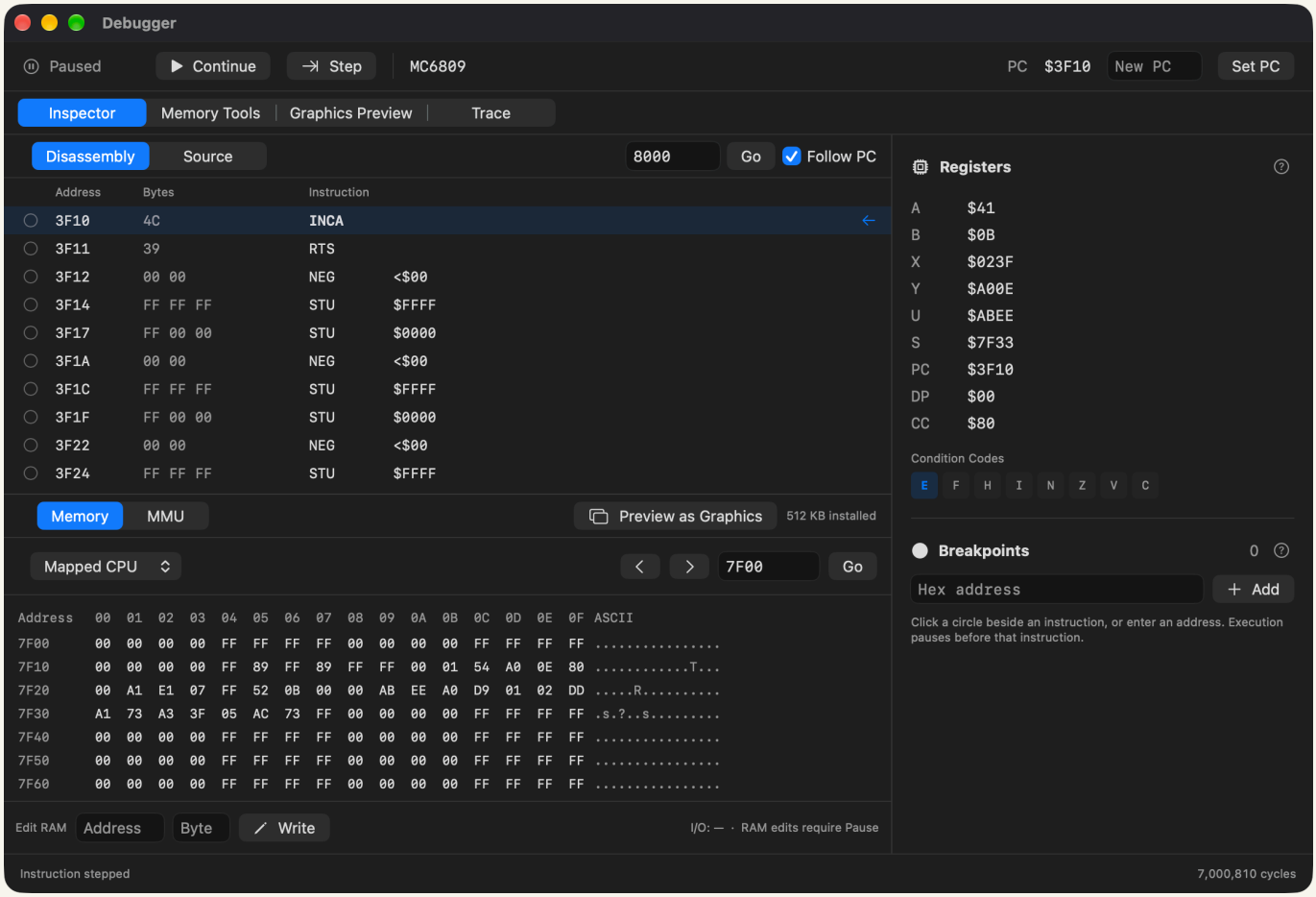


Figure 5: The debugger shows the saved return address in guest RAM during the call to nextletter. Read the two bytes at S in order, 3F then 05.

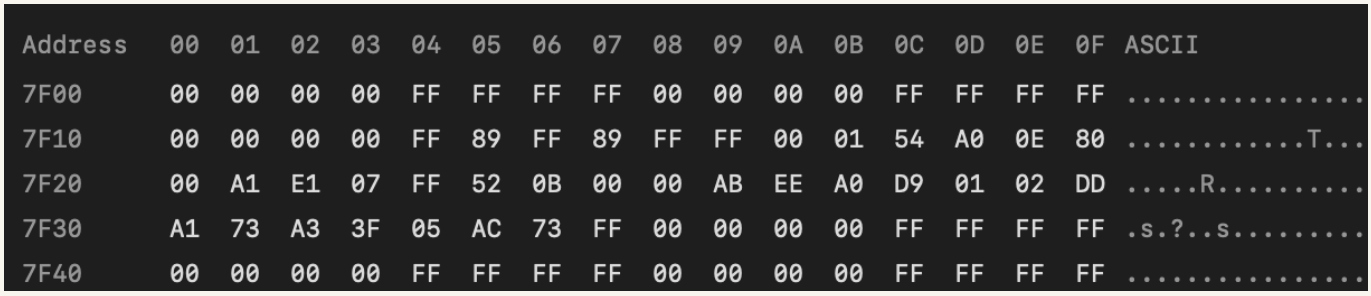


Figure 6: In the magnified memory grid, row 7F30 contains 3F under column 03 and 05 under column 04.

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**.

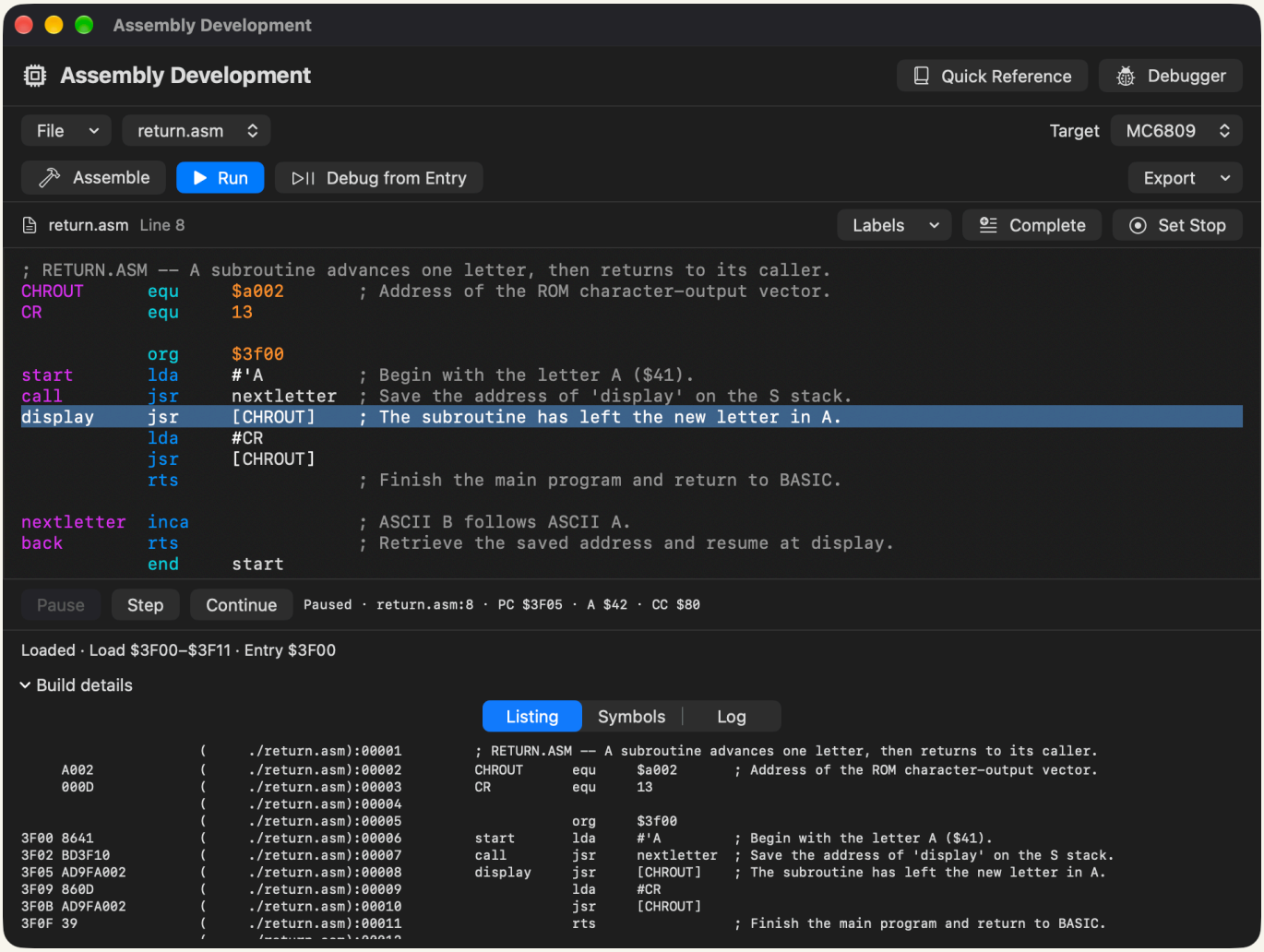


Figure 7: After the helper returns, display is highlighted with PC $3F05 and A $42. The listing below shows the call at $3F02 followed by display at $3F05.

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.

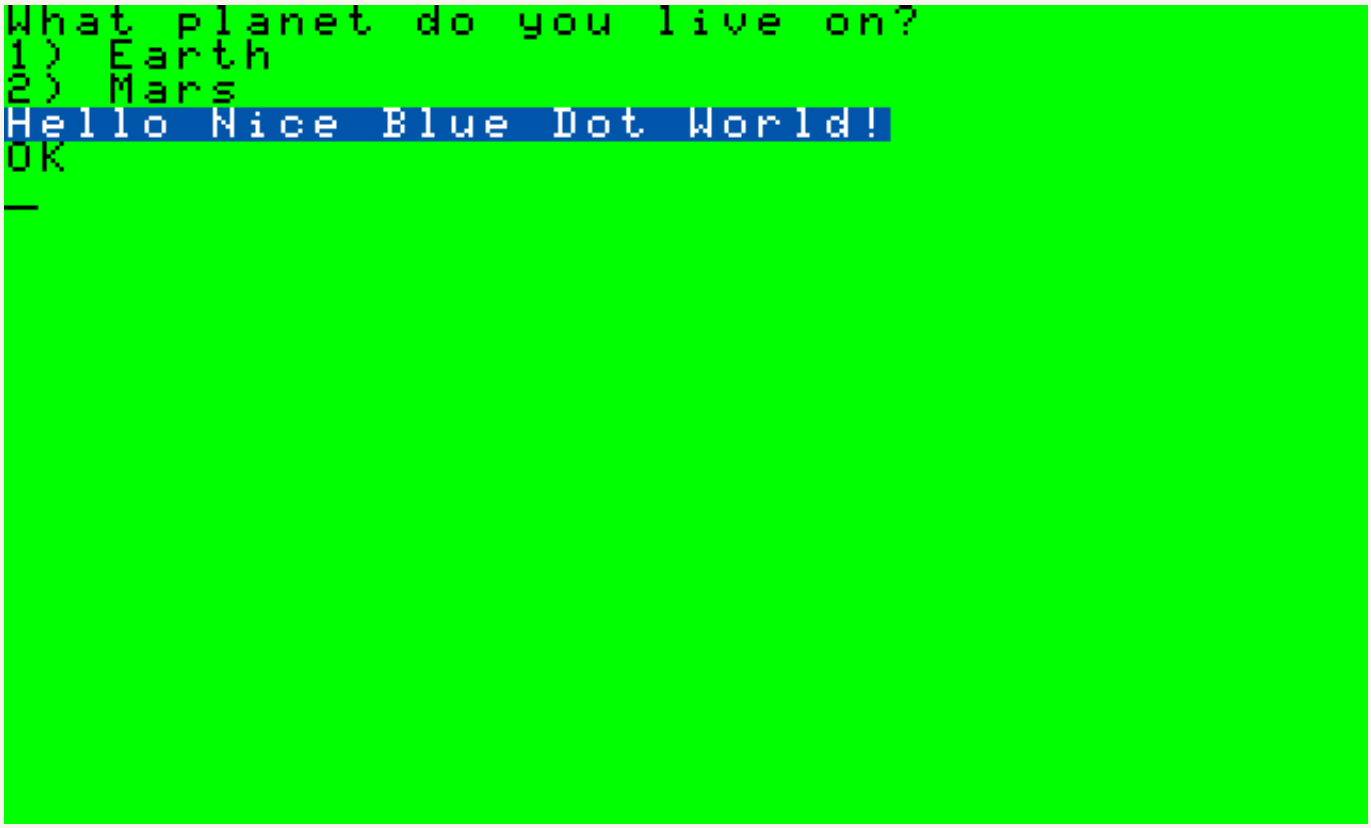


Figure 8: The CoCo 3 shows the two planet choices, then Hello Nice Blue Dot World! on blue after key 1.



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

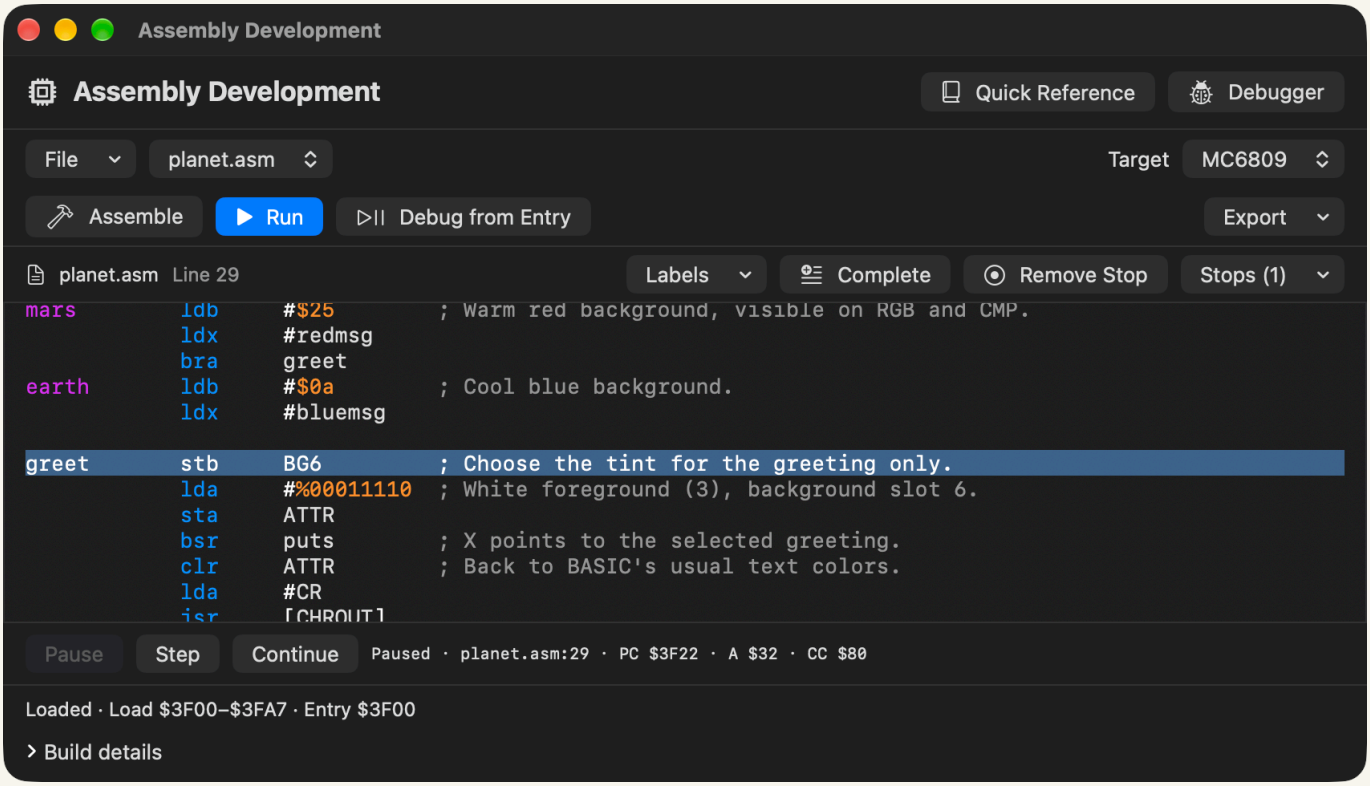


Figure 10: The assembly debugger stopped at greet. The next instruction will store register B in the background palette slot.

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.

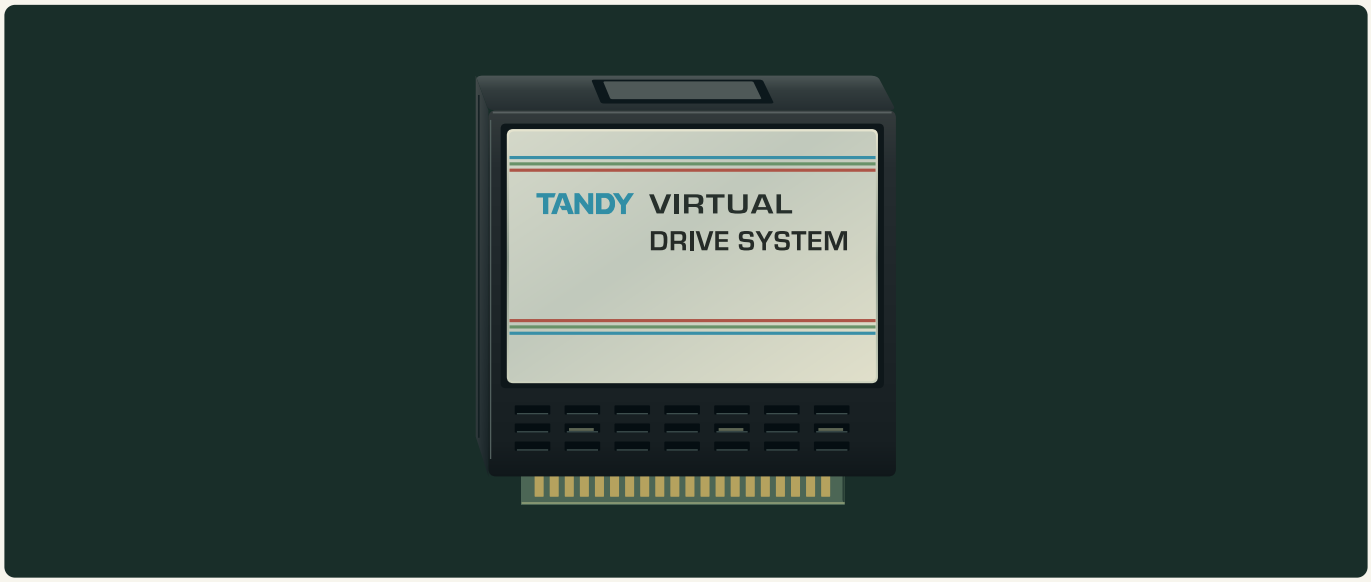
The CoCo 3 and CM-8 monitor presentation in VCC Plus's skeuomorphic System view was inspired by SockMaster's "pic of CoCo 3 and CM-8 monitor," published on CoCopedia. VCC Plus's artwork was designed and rendered independently. The remaining hardware illustrations were independently designed using historical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

Read this book online

CMOC license notice · LWTools COPYING notice

NitrOS-9 on VCC Plus — A Practical Quick Start

Leave a captain's log on a disk, start a fresh session, and find your message again. Meet the shell, follow a path through directories, and discover which computer is listening.



VCC Plus illustration of the dark Virtual Drive System cartridge, its silver nameplate and gold edge connector.

Brian K. Holman

Book 8 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

A different desk on the same CoCo 3
Find the disk behind the name 6
Leave a captain's log 9
Find your way back 12
Return to familiar BASIC 15
A connection to another computer 17
At the OS-9 prompt 20
Publication and artwork notices 23

A different desk on the same CoCo

Chapter 1 of 6

The familiar green screen is one way to talk to your Color Computer. Today you will visit another. NitrOS-9 can keep several programs at work, organize files in directories and give you a shell where you can start programs by name. Your first mission is small: leave a captain's log on a disk, come back in a fresh session and find it again.

You already know how to choose a VCC Plus profile and save a BASIC program. If those steps are new, start with Back at the CoCo. This visit uses the supplied **NitrOS-9 L2 V3.3.0** setup. The book works offline with the installed profile's prepared media.

Make a practice setup

Save any program, disk changes or cassette recording you want to keep before changing profiles. Loading a profile restarts the emulated computer. A saved profile remembers settings and file locations; it does not preserve the contents of RAM.

For this book, make your own fresh media. That gives the log a clean home and keeps the files in your everyday setup separate.

1. Choose **Machine** → **Configuration Profiles…**. Select **NitrOS-9 L2 V3.3.0** under **System Profiles**.
2. Under **Make it yours**, choose **Create Fresh Starter Media…**. Give the personal profile a recognizable name, such as **Mission Lab**, then choose **Save**.
3. Select that copy under **My Profiles**, choose **Load…**, and confirm **Load Profile** after reviewing the change.
4. The personal profile loads paused. Close the profile window and choose **Run** in the emulator.

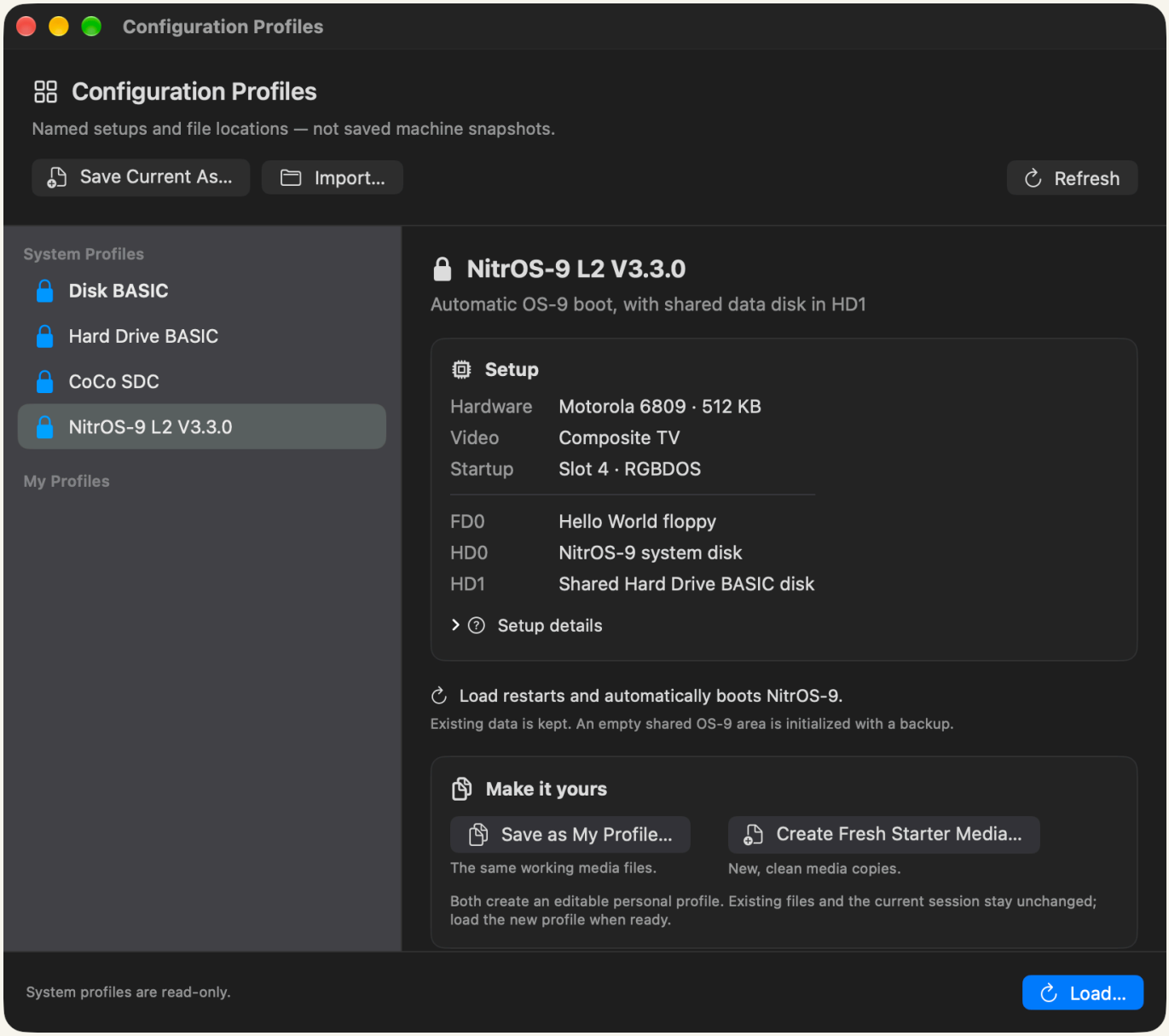


Figure 1: Configuration Profiles shows the locked NitrOS-9 setup. Under Make it yours, Create Fresh Starter Media makes a separate practice set.

Copying a system profile and loading it are separate actions. **Save as My Profile...** keeps references to the same working media; **Create Fresh Starter Media...** supplies separate clean copies. Choose the fresh-media action for this exercise. Keep the resulting personal profile for the whole book. Creating another fresh copy later would give you another empty data disk.

The setup uses **Motorola 6809**, **512 KB** and **Composite TV**. The examples were also checked on the emulated HD6309, but you do not need to change processors. A source build without the privately supplied ROMs and system image can show an unavailable-profile explanation. Use a build with the prepared media to follow the lesson.

Wait for your new prompt

VCC Plus enters the boot command for this profile. Leave the keyboard alone while the system starts. Eventually a prompt appears ending in **/DD:**. The beginning of the prompt may include a terminal name and process number, such as **{Term|02}**.

A **shell** reads a command, finds the program that carries it out, and presents another prompt when that command finishes. Here the shell is **Shell+**, running under NitrOS-9. You can type lowercase commands. There are no BASIC line numbers before them.

Where you are	Who handles the request?	A clue
Configuration Profiles or a Mac menu	The VCC Plus application on your Mac	Native windows and buttons
The BASIC screen	The CoCo's BASIC interpreter	**OK** after a command
The NitrOS-9 command line	The guest OS and its shell	A prompt ending in a directory such as **/DD:**

The same emulated CPU and keyboard serve both BASIC and NitrOS-9. Different software is now receiving the keys. BASIC's `PRINT` command is not a shell command. We will deliberately return to BASIC in chapter 5, so you can see the change yourself.

Locate the two working images

Open **Media → Virtual Hard Disks...**. Read the assignments for **HD0** and **HD1** before writing anything. In this practice copy, HD0 is the prepared NitrOS-9 system image and HD1 is the fresh data image. Note their filenames. In Configuration Profiles, the personal profile's **Files** list shows the locations; two copies can have the same filename in different folders. Keep the images writable with automatic saving for this lesson. Close the controls, choose **Run** if the guest is paused, and click the CoCo display before typing.

The built-in system profile normally shares its HD1 working file with **Hard Drive BASIC**. Your fresh-media copy has its own files. Inside HD1, the OS-9 filesystem and the BASIC disk volumes occupy separate areas. The lesson writes only to the OS-9 area.

An operating system supplies a way to name devices and find files. The next chapter connects **HD0** and **HD1** in the Mac window to the names you will type at the shell.

Find the disk behind the name

Chapter 2 of 6

The prompt ends in **/DD:**. In the Mac's Virtual Hard Disks window, the drives are called **HD0** and **HD1**. Why do the names disagree? Each belongs to a different layer of your computer. The Mac attaches an image file to a drive; OS-9 gives a name to the filesystem it opens there.

Ask three names for their directories

Click the CoCo display. At the ready shell prompt, enter these commands separately, pressing Return after each one:

```
dir /dd
dir /h0
dir /h1
```

In the supplied profile, **/dd** and **/h0** show the same system directory. Look for names such as **CMDS**, **GAMES** and **SOURCES**. **/h1** opens the data filesystem on the second hard disk. On fresh practice media, its heading is followed by the next prompt: there are no files there yet.

Name	In this profile
HD0, in the Mac controls	The image containing the prepared NitrOS-9 system
/dd and **/h0**, in the shell	Two device names reaching that system filesystem
HD1, in the Mac controls	The working data image
/h1, in the shell	The OS-9 filesystem in the working data image

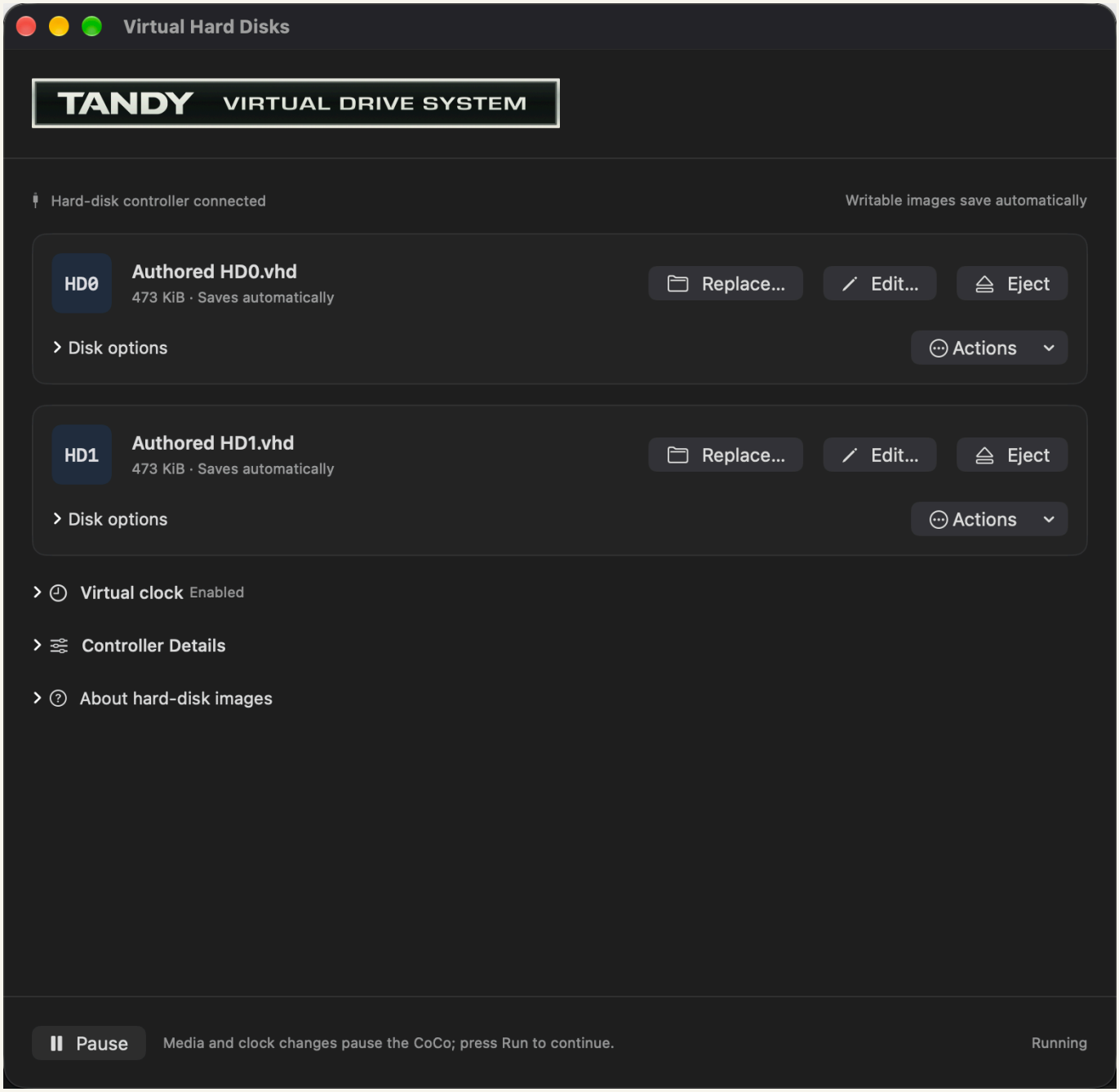


Figure 2: The current Virtual Hard Disks window shows separate HD0 and HD1 assignments, save modes and Edit buttons using small authored examples. Your practice profile uses its NitrOS-9 system image in HD0 and working data image in HD1; check those filenames in your own window.

Keep the **in this profile** qualification. Device names belong to a configured OS-9 system. Another boot disk can use a different arrangement.

If **/h1** already contains files, keep them. You may have opened your existing working media instead of a new practice copy. The log lesson needs one unused directory name. It does not need an empty hard disk.

Read a path from left to right

Here is the destination you will create in the next chapter:

```
/h1/mission/captains.log
```

Part	Job
The first **/**	Start with a device name, rather than the current directory
h1	Choose the data filesystem
mission	Enter a directory on it
captains.log	Name a file in that directory

The slashes separate the steps. This is an *absolute path*: it gives the route from the device to the file. In historical OS-9 documentation, you will also see the word *pathlist*.

A shorter name such as **captains.log** is relative to your current data directory. It becomes useful after you have moved into **mission**. That will save typing, but it also makes your current location important. A perfectly good filename can produce an error when you ask for it in the wrong place.

OS-9 file lookup here ignores letter case. The prompt may show **/H1** while you type **/h1**. Those names reach the same device. Text stored inside a file keeps its own case; **beacon in sight** will stay lowercase.

Keep BASIC and OS-9 files apart

The working data image supplied with the system profiles contains separate space for OS-9 files and RGB-DOS BASIC volumes. That explains something that can look alarming on a first visit: **/h1** can be empty even though you remember a HELLO program on that image.

HELLO belongs to the BASIC part. An OS-9 directory lists the OS-9 part. Neither listing is a catalog of every byte in the Mac image file.

The default floppy is also a BASIC disk. Attaching it to FD0 does not turn it into an OS-9 filesystem. Keep it as it is. This book uses the already prepared hard disks, so no formatting is required. Software, Storage, and Peripherals follows the same working image from the BASIC side.

The Mac's **Disk Image Editor** can browse supported raw OS-9 and Disk BASIC filesystems without booting them. Choose **Edit…** beside HD0 or HD1 in **Virtual Hard Disks…**. The mounted machine pauses and stays locked while the editor is open; applying edits requires a restart. In an image shared by OS-9 and HDB-DOS, the volume selection distinguishes the OS-9 filesystem from its numbered BASIC disks. Selecting one does not show the files in the others, and the editor's volume names do not replace guest names such as **/h1**. A numbered BASIC disk that overlaps declared OS-9 boot data is protected and omitted from the editable choices. For these lessons, continue with the guest commands below; follow the native editor guide when you want to inspect a working copy from the Mac.

Give commands their own home

OS-9 keeps a current **data directory** for your files and a current **execution directory** for finding commands. The prepared system puts its commands in **CMDS**. Moving your data directory to **/h1/mission** will let you use a short log filename while still running commands such as **dir** and **list**.

Think of the data directory as your desk and the execution directory as the tool cupboard. You can move your papers to a different desk without carrying the cupboard along. We will change only the data directory in this lesson.

The next chapter turns those names into something worth finding again: your first captain's log.

Leave a captain's log

Chapter 3 of 6

The beacon is in sight. You want to leave a message that will survive after this CoCo session ends. A line on the screen can disappear; a file on your working disk can wait for you.

Make a place for the message

Use the fresh practice media from chapter 1. At the OS-9 shell prompt, type:

```
makdir /h1/mission
```

makdir makes a directory. The path chooses the OS-9 data filesystem on HD1 and names the new directory **mission**. Now move into it:

```
chd /h1/mission
pwd
```

chd changes your current data directory. **pwd** prints that directory's path. Look for **/H1/mission** in the reply and at the end of the prompt. You now have two ways to check where your short filenames will go.

If making the directory reports **ERROR #218**, the name already exists. Perhaps you tried the lesson before. You can use **chd /h1/mission** to visit that directory and **dir** to inspect it. Keep any file you find. For a completely separate repeat, choose a different directory name and use it consistently in the following commands.

Write one line, then read it

Enter this command exactly, with a space before **>** and no space inside the filename:

```
echo beacon in sight >captains.log
```

echo produces the words **beacon in sight**. The **>** asks the shell to send that output into a new file. You have redirected the output from the screen to **captains.log**. Since you are in **/H1/mission**, that short filename means **/h1/mission/captains.log**.

The prompt should return. Check the directory, then read the message:

```
dir
list captains.log
```

dir includes **captains.log**. **list** displays:

```
beacon in sight
```

```
{Term!02}/DD:dir /h1

 Directory of /h1  2026/11/06 18:51


{Term!02}/DD:makdir /h1/mission

{Term!02}/DD:chd /h1/mission

{Term!02}/H1/mission:pwd
/H1/mission

{Term!02}/H1/mission:echo beacon in sight >captains.log

{Term!02}/H1/mission:dir

 Directory of .  2026/11/06 18:51
captains.log

{Term!02}/H1/mission:list captains.log
beacon in sight

{Term!02}/H1/mission:
```

Figure 3: The running CoCo creates /h1/mission, changes into it, and prints the path with pwd. After echo writes captains.log, dir lists the file and list reads back beacon in sight.

You have used several small programs together. One made a directory, one reported your location, one produced a line of text, and another read the file. The shell connected the text producer to a file through **>**. That ability to combine tools is one of the pleasures of an operating system.

If the log already exists

With this supplied shell, repeating the same **echo … >captains.log** reports **ERROR #218** and leaves the existing file intact. Read it with **list captains.log**. To make a second log, choose a new filename. We will leave overwriting, appending and deleting for later lessons.

Make sure the Mac can keep it

OS-9 has finished its command when the prompt returns. VCC Plus also needs to save the changed image to its backing file on the Mac.

Open **Media → Virtual Hard Disks…** and inspect **HD1**. Your practice image should be writable and report **Saves automatically**. Expand **Disk options** if needed: **Write protected** and **Temporary changes** should both be off for this exercise. Let **Saving automatically…** finish before leaving the session.

If the row shows **Temporary changes**, your new bytes are being kept in memory. Use **Save As…** to retain a copy, or deliberately turn off Temporary changes to allow pending changes to save to the assigned file. If you used **Save As…**, update your personal profile's HD1 file reference to the saved copy before reloading it. The saved profile still points to the earlier file until you update it. Save As retains the temporary-changes mode. If saving reports an error, resolve it before continuing.

There are now three places you can identify:

Place	What belongs there
The current CoCo display	The commands and replies you just saw
/h1/mission/captains.log	The text file OS-9 created
The Mac image assigned to **HD1**	The saved disk containing that filesystem and its files

Write down the HD1 filename or keep the personal profile created for this practice session. Where is my work? explains what a profile remembers. The next chapter gives the file a stronger test: finding it after a fresh start.

Find your way back

Chapter 4 of 6

Your last screen showed **beacon in sight**. That proves the file could be read in that session. Now give the CoCo a fresh start and find the message from its saved disk.

Bring back the same working image

Finish the previous chapter's save check first. The OS-9 command must have returned, and HD1 must have finished saving to its assigned Mac image.

Quit VCC Plus, reopen it, and load the **same personal practice profile** you made for this book. Do not create another fresh set of starter media for this step. A fresh set would contain a fresh, empty data filesystem. Loading your saved profile brings back its existing media assignments while starting a new guest session. If VCC Plus is paused, click **Run** and let NitrOS-9 boot.

Open **Media → Virtual Hard Disks…** and confirm that HD1 has the same working image. Close the drive window and click the CoCo display. Wait for the shell prompt ending in **/DD:**. Your current directory has returned to the system disk; it is no longer **/H1/mission**.

A profile is useful here because it remembers the setup. The message itself lives in the disk image. Saving a profile and saving a changed disk solve different problems.

To recover an entire practice disk and prove which version you have, see Restore the right copy.

Take one deliberate wrong turn

At the fresh shell prompt, enter:

```
list /h1/captains.log
```

The supplied system reports **ERROR #216**. That can be an unsettling greeting after a restart, but examine the path. You asked for **captains.log** directly in the top directory of **/h1**. You created it inside **mission**.

This error is evidence about the path you requested. It is not proof that the file disappeared. The next step is to inspect the directory where you put it.

Follow the complete path

Enter:

```
dir /h1/mission
list /h1/mission/captains.log
```

The first command lists **captains.log**. The second reads:

```
beacon in sight
```

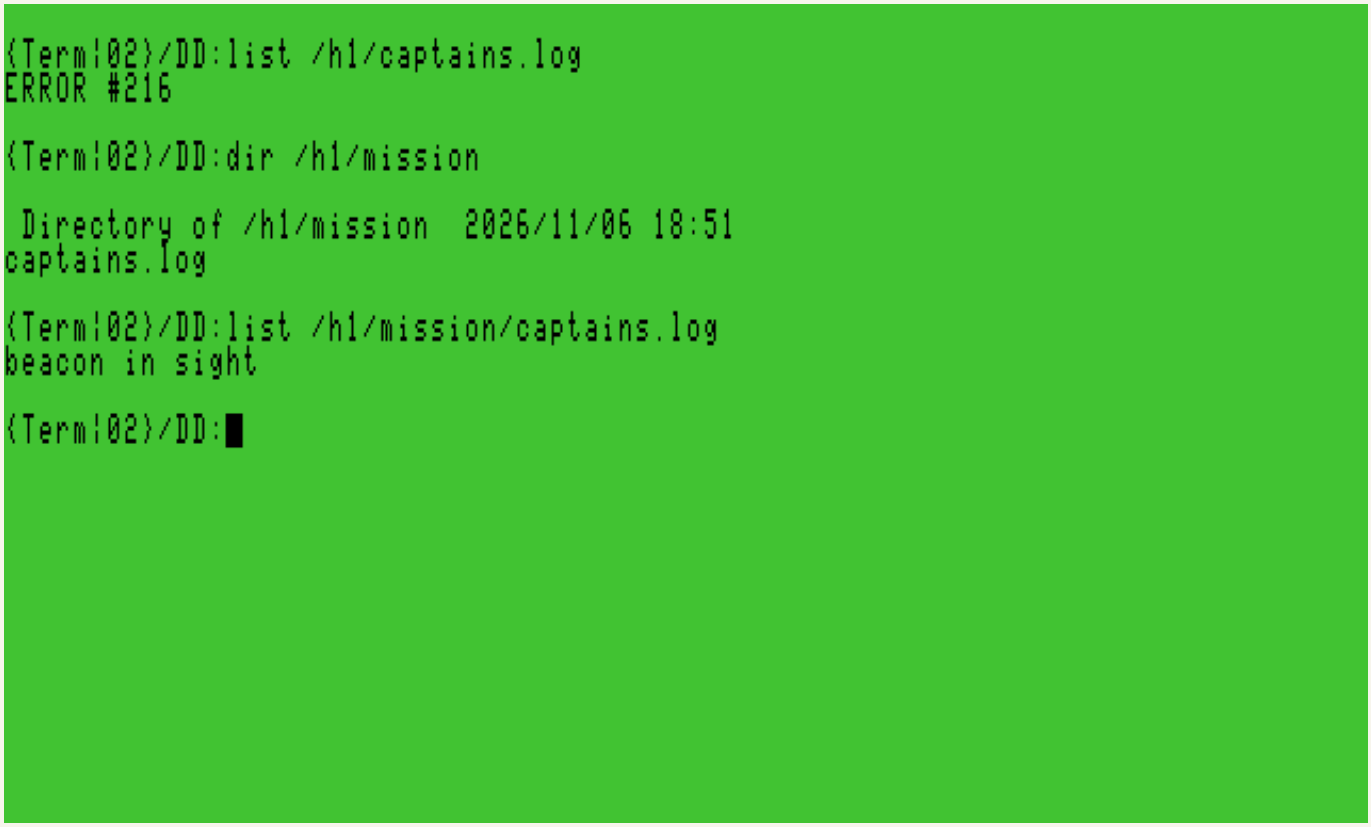


Figure 4: A fresh guest reports error 216 for /h1/captains.log. The directory /h1/mission contains captains.log, and its full path reads the saved message, beacon in sight.

The prompt still ends in **/DD:**. You did not need to move into **mission** to read a file there: the full path gave **list** everything it needed. Compare that with the short name you used while working inside the directory.

Your current directory	Name supplied to list	Result for this exercise
/H1/mission	**captains.log**	Finds the log relative to your current directory
/DD	**/h1/captains.log**	Looks in the wrong directory; error 216
/DD	**/h1/mission/captains.log**	Follows the complete path and reads the log

You have now recovered a file from saved storage. The old guest's RAM is gone, but the working disk supplies the message again.

When your result differs

Start with the smallest question that explains the difference.

What you see	What to check
/h1/mission cannot be found	Is HD1 the same working image used in chapter 3? Did you choose a different directory name?
The directory exists, but the log is missing	Use **dir /h1/mission** to inspect its actual filenames. Check whether you chose a different log name.
HD1 reports unsaved changes or a save error	Keep that session and its data. Follow the save message before replacing or ejecting the image.

What you see	What to check
You created a new set of starter media	Load the earlier personal practice profile, or reattach its saved working image.
You see BASIC's **OK**, not an OS-9 shell	Load the practice profile, choose **Run**, and wait for NitrOS-9 to boot.

Avoid formatting a disk to solve a missing-file problem. Formatting prepares a filesystem; it does not search for your old message. The useful clues are the assigned Mac image, the directory path and the filename.

Keep your log. In the next part of the journey, you will visit BASIC and then come back to the same disk.

Return to familiar BASIC

Chapter 5 of 6

Your log is safely on disk, and the shell is waiting for another command. Now leave NitrOS-9, ask BASIC a tiny question, and return. This round trip makes an important distinction visible: restarting a computer changes the running software, while a saved file remains on its disk.

Go through the archive's menu

Finish the current command and let automatic disk saving complete. Save work in any other guest programs before restarting. Then use the emulator toolbar's **Restart → Cold Restart – Clear RAM...** and confirm **Cold Restart**.

The supplied system image has its own startup menu. After this restart you see **R.Gault's Emulator Hard Drive System**, rather than an ordinary BASIC command prompt. Its sixth choice is **EXIT TO BASIC**. Type **6** and press Return.



Figure 5: The supplied hard-drive menu after Cold Restart. Its sixth choice is EXIT TO BASIC.

When **HARD DISK BASIC** and **OK** appear, type this command and press Return:

```
PRINT 2+3
```

BASIC prints **5**, then **OK**. You are still using the hard-drive setup with RGBDOS. Cold Restart kept the mounted image files and cleared the emulated RAM. Choosing the exit item completed this image's route back to BASIC.

Boot the operating system again

At the BASIC prompt enter:

```
DOS 255
```

Wait for the shell prompt ending in **/DD:**. Then enter:

```
list /h1/mission/captains.log
```

The answer is still **beacon in sight**. BASIC and NitrOS-9 took turns running on the same computer, and the file survived because it was saved to the image that remains in HD1.

In this supplied setup, **255** identifies the BASIC hard-disk volume containing the OS-9 boot information. It is not the number of a physical drive attached to the Mac. Other images may boot differently; keep this command paired with the profile used in the book.

Change back to your ordinary setup

If you want the familiar floppy-based environment, open **Machine → Configuration Profiles...**, select **Disk BASIC**, and choose **Load...**. Read and confirm the restart. This changes the equipment and media assignments as well as starting BASIC again. Your Mission Lab personal profile still points to its own working images.

Later, load **Mission Lab** and choose **Run** to boot its OS-9 system again. The profile gives you the settings and disks. The saved file on HD1 gives you the message. Those are two separate pieces of the journey.

For the differences between Pause, Soft Reset and Cold Restart, see the User's Handbook. Before you leave the quick start, there is one optional visit to a different computer: the Mac that is hosting this CoCo.

A connection to another computer

Chapter 6 of 6

You have used a shell inside the emulated CoCo. Your Mac has a shell too. VCC Plus can let an OS-9 terminal program talk to it, while keeping the two machines' files and permissions distinct.

This chapter is optional. It needs no public server, account or internet service. Keep this first visit to identifying the host and leaving it.

Open the two gates

Start at the OS-9 shell. If you switched to Disk BASIC at the end of chapter 5, load **Mission Lab**, choose **Run**, and wait for **/DD:**.

Open **Machine → Machine Settings…** and find **Enable DriveWire networking**. The supplied NitrOS-9 system disk loads the required guest drivers during boot. For **Mission Lab**, turn the switch on if it is off. A personal profile does not turn on networking by itself. The built-in NitrOS-9 profile defaults to on, and VCC Plus honors your explicit choice across profiles.

That switch allows supported guest software to open outbound TCP connections. The **Mac shell** has a separate permission because it runs commands as your Mac user. Those commands can read or change real Mac files. The shell starts in your Mac home folder and uses your normal shell startup files.

Return to the CoCo, resume if paused, click its display, and type:

```
telnet macshell 2323
```

The name **macshell** identifies VCC Plus's guest-only connection. It does not publish a shell for other computers to connect to. A permission request appears in the Mac interface. Read it and choose **Allow** for this visit. You can choose **Deny** and skip the exercise instead.

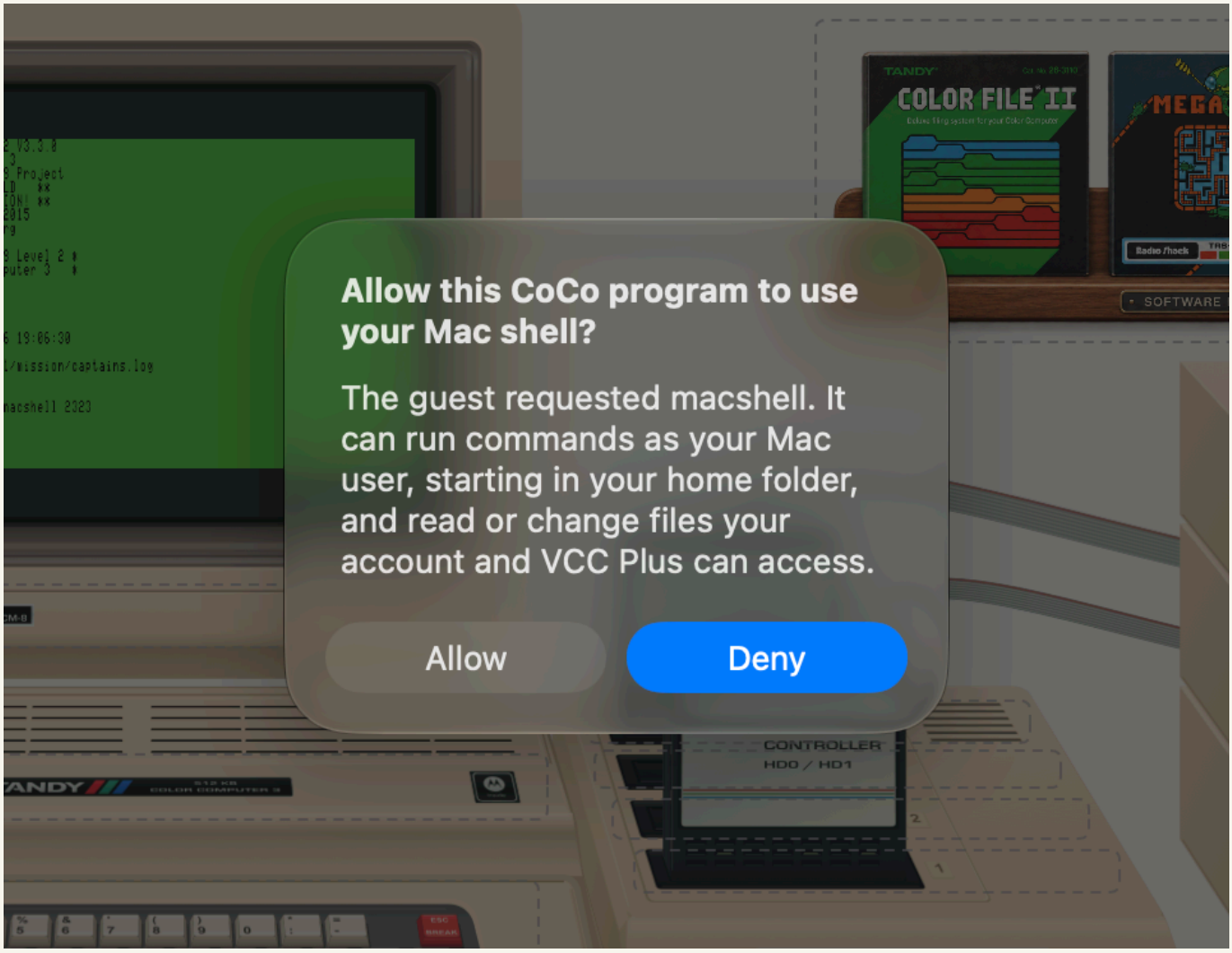


Figure 6: VCC Plus asks whether to let the guest use the Mac shell. Deny declines the request; Allow admits this one connection from the practice setup.

On the personal Mission Lab setup, **Allow** is for this connection. Only a recognized built-in NitrOS-9 setup can offer a remembered grant. That grant applies to guest programs using that setup. Machine Settings also offers **Start Mac Shell Session** to prepare one guest connection explicitly.

Ask which computer answered

At the new Mac shell prompt, type the following and press Return:

```
/usr/bin/uname
```

The reply includes **Darwin**, the name reported by the Mac's underlying operating system. The prompt itself depends on your shell and startup files; it may look different from someone else's. You are now issuing a Mac command through the CoCo's terminal display.

The directory **/h1/mission** you used earlier belongs inside an emulated disk image. It is not the same as a directory in the Mac shell. Pause to read the prompt whenever you change connections. The same keyboard can be addressing a different filesystem.

For a second console or a graphics window inside the CoCo, see the OS-9 terminal lesson.

Close the connection

Type:

```
exit
```

The connection closes and the OS-9 prompt returns. If you need to end the session from the Mac side, use **Stop Mac Shell Session** in Machine Settings. **Forget Mac Shell Permission** removes a remembered grant. Turning networking off closes active connections too.

The short identification-and-exit sequence was checked on both guest processors. More elaborate terminal programs can depend on the guest client's key handling and display capabilities. This quick start does not require an interactive Mac editor.

For the controls in one place, see Let OS-9 connect. For the commands and device names used throughout this book, turn to the field guide. You can now find a disk, make a file, recover it after a restart, and tell which computer is listening. Those are useful foundations for the deeper OS-9 journey ahead.

At the OS-9 prompt

Appendix A

Keep this page beside the emulator while you explore. Begin by looking at the prompt. Commands belong to the computer that is listening: BASIC, the OS-9 shell, or an explicitly opened Mac shell.

Which prompt is this?

What you see	Who is listening	What to do next
A Shell+ prompt ending in `/DD:` or `/H1/mission:`	The CoCo's OS-9 shell	Try the directory and file commands below. The path names the current data directory.
`OK` after the HARD DISK BASIC banner	CoCo BASIC with RGBDOS	`PRINT 2+3` prints `5`. `DOS 255` starts the supplied NitrOS-9 system again.
R.Gault's **Emulator Hard Drive System** menu	The archive's BASIC menu program	Choose **6**, **EXIT TO BASIC**, and press Return. This is the extra step after the cold restart.
Your Mac's shell prompt after allowing `telnet macshell 2323`	Your Mac user account	`/usr/bin/uname` prints `Darwin`. `exit` ends that shell and returns to OS-9.

A Mac shell prompt can vary with your own settings. Recognize it by the connection you deliberately opened, rather than by expecting a particular symbol. See An optional connection to another computer.

Find a disk, directory or file

Type one command, press Return, and wait for the next prompt before entering another. These examples use the supplied NitrOS-9 L2 V3.3.0 setup and the log created in Leave a captain's log.

Your task	Command	Checkpoint
Look at the system disk	`dir /dd`	The listing includes `CMDS`, `GAMES` and `SOURCES`.
Use its other device name	`dir /h0`	You see the same system directory.
Look at the data filesystem	`dir /h1`	A fresh working copy is empty. Keep any files already present on your own copy.
Find the log's directory	`dir /h1/mission`	The listing includes `captains.log` after the exercise.
Work inside that directory	`chd /h1/mission`	The prompt's path changes to `/H1/mission:`.
Check your current data directory	`pwd`	It prints `/H1/mission`.
Read the file from that directory	`list captains.log`	It prints `beacon in sight`.
Read the file from another directory	`list /h1/mission/captains.log`	The full path reaches the same saved message.

An initial `/` makes the path start with a device name. In `/h1/mission/captains.log`, `/h1` names the data filesystem, `mission` names its directory, and `captains.log` names the file. Without that initial device path, OS-9 looks for a data file relative to your current data directory.

OS-9 file lookup is case-insensitive in this setup. The screen can display `/H1/mission` even when you typed `/h1/mission`. Text *inside* a file keeps its own case. The command name and its arguments also do different jobs: `list` is the program to run; `captains.log` is the data it reads. The shell has a separate execution directory for finding command programs. See Find the disk behind the name before assuming a command must live beside your log.

Leave a new log

Use these commands once on a fresh practice copy where `mission` does not yet exist:

```
makdir /h1/mission
chd /h1/mission
pwd
echo beacon in sight >captains.log
dir
list captains.log
```

`makdir` creates the directory. `chd` changes where the following relative filename will be found. The `>` sends the output of `echo` into a new file. `list` reads those saved characters back.

The output should finish with **beacon in sight**. Let the command complete and the app finish saving the writable hard-disk image. Preserve that same HD1 image when you restart. Find your way back shows how to recover the log from a new session.

When a command answers with an error

Message in this exercise	What happened	Useful next step
ERROR #216 after `list /h1/captains.log`	That path does not name the saved file. The log is inside `mission`.	Enter `dir /h1/mission`, then `list /h1/mission/captains.log`.
ERROR #218 after repeating `makdir /h1/mission`	The directory already exists.	Use `chd /h1/mission` to enter it and `dir` to inspect it.
ERROR #218 after repeating `echo beacon in sight >captains.log`	This shell's single `>` would create a file whose name already exists. The tested command leaves the original intact.	Read it with `list captains.log`, or repeat the lesson using a new filename.

An error number is a clue about the command that just ran. Check the path and the current directory before deciding that a disk or file has disappeared. A fresh practice copy is useful for repeating the lesson; it does not require replacing the disk that holds your earlier work.

Map the equipment to the names

The following map describes the built-in **NitrOS-9 L2 V3.3.0** profile. Fresh-media copies use their own files. Inspect the mounted assignments in VCC Plus to identify the actual Mac files for your session.

Mac image or component	VCC Plus assignment	Meaning inside this setup
Working `NitrOS-9 L2 V3.3.0.vhd`	**HD0**, through the hard-disk controller in Multi-Pak slot 1	`/dd` and `/h0` reach its OS-9 system filesystem.
Working `Hello World.vhd`	**HD1**, through the same controller in slot 1	`/h1` reaches its OS-9 data filesystem. This is where the captain's log goes.

Mac image or component	VCC Plus assignment	Meaning inside this setup
Working `Hello World.dsk`	**FD0**, through the FD-502 controller in slot 4	The supplied floppy holds a BASIC filesystem. It is not an OS-9 practice disk.
RGBDOS ROM	FD-502 in startup slot 4	BASIC can boot the supplied system with `DOS 255`. BASIC volume numbers and the app's HD0/HD1 labels describe different levels of storage.

The shared `Hello World.vhd` has separate OS-9 and BASIC areas. The ordinary **Hard Drive BASIC** profile mounts that shared working file as **HD0**. The NitrOS-9 profile mounts it as **HD1**. Changing the app's assignment does not turn its BASIC files into OS-9 files. See Which part of the image? for the broader storage map.

The tested starting machine uses the **MC6809**, **512 KB**, and **Composite** video. These exercises also passed on the **HD6309** in its startup mode. A profile saves equipment choices and file references. It does not preserve a running process or its memory. Where is my work? explains how to keep those different kinds of state straight.

See which guest drivers serve these devices in Trace three paths from the same shell.

Continue the journey

For the surrounding Mac controls, use The VCC Plus User's Handbook. For media formats and the difference between a mounted image and the files inside it, turn to Software, Storage, and Peripherals.

Take the next watch in Working with OS-9. Turn the log into a shell procedure, follow its bytes onto disk, or assemble a report writer inside OS-9. Keep the command finder handy while you work.

Books behind the journey

- The NitrOS-9 Project. *NitrOS-9 Operating System User's Guide*. Version 03.02.01, 2004. Sections 2.1–2.3, pp. 5–7, on the shell; sections 3.2–3.5, pp. 10–14, on devices, pathlists and directories; section 3.8, pp. 15–16, on working directories; section 4.2.2, pp. 23–24, on redirection.
- Tandy Corporation and Microware Systems Corporation. *OS-9 Commands Reference*. 1986. CHD/CHX, p. 6-12; DIR, pp. 6-35–6-37; ECHO, p. 6-42; LIST, p. 6-59; MAKDIR, p. 6-63; PWD/PXD, p. 6-82.
- Tandy Corporation. *Getting Started With OS-9*, in *OS-9 Level Two Operating System*. 1986. Chapter 4, pp. 4-5–4-6, on pathlists and device names; chapter 6, pp. 6-1–6-5, on directories and files.

The Color Computer Archive preserves historical documentation that informed this book. The lessons and command exercises 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

Assembly Language Programming for the VCC Plus CoCo 3

A receiver reports three letters. Follow them through the processor, the stack and RAM. Then make the receiver fail on purpose and prove where it took the other branch.



The VCC Plus Hitachi processor badge.

Brian K. Holman

Book 9 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Read the packet . . . 3
Give the receiver a damaged signal . . . 6
Give a subroutine a contract . . . 7
Borrow a different byte at the same address . . . 9
Draw on the page you cannot see . . . 12
Move the packet with the HD6309 . . . 15
Read the return ticket an interrupt leaves . . . 17
Deliver the receiver . . . 19
Instruction directory for the 6809 and 6309 . . . 22
Registers, addressing and LWASM . . . 69
Keep a workbench notebook . . . 85
Publication and artwork notices . . . 87

Read the packet

Chapter 1 of 8

A receiver reports **ORB**. Did it print those three letters from the message it received, or were they just the programmer's favorite three letters? Catch the receiver after it has made a copy and before it decides what to print. At that instant you can ask the machine for evidence.

Set a stop on a decision

Start with a fresh **Disk BASIC** setup. Save any guest work before loading a program. Save dispatch.asm, choose **Develop → Assembly Development...**, then **File → Open Source...** and open your copy. Choose **MC6809** as the assembler target. This program also runs on an HD6309. If stepping is new to you, try the one-byte experiment first.

Choose **Assemble**, open **Build details → Listing**, and look for three labels: **packet**, **received** and **inspect**. A label names the current assembly address. It can mark data just as easily as an instruction. Select the CMPA instruction on the **inspect** line, choose **Set Stop**, then **Run**. Confirm the load. Leave the machine paused for the next section.

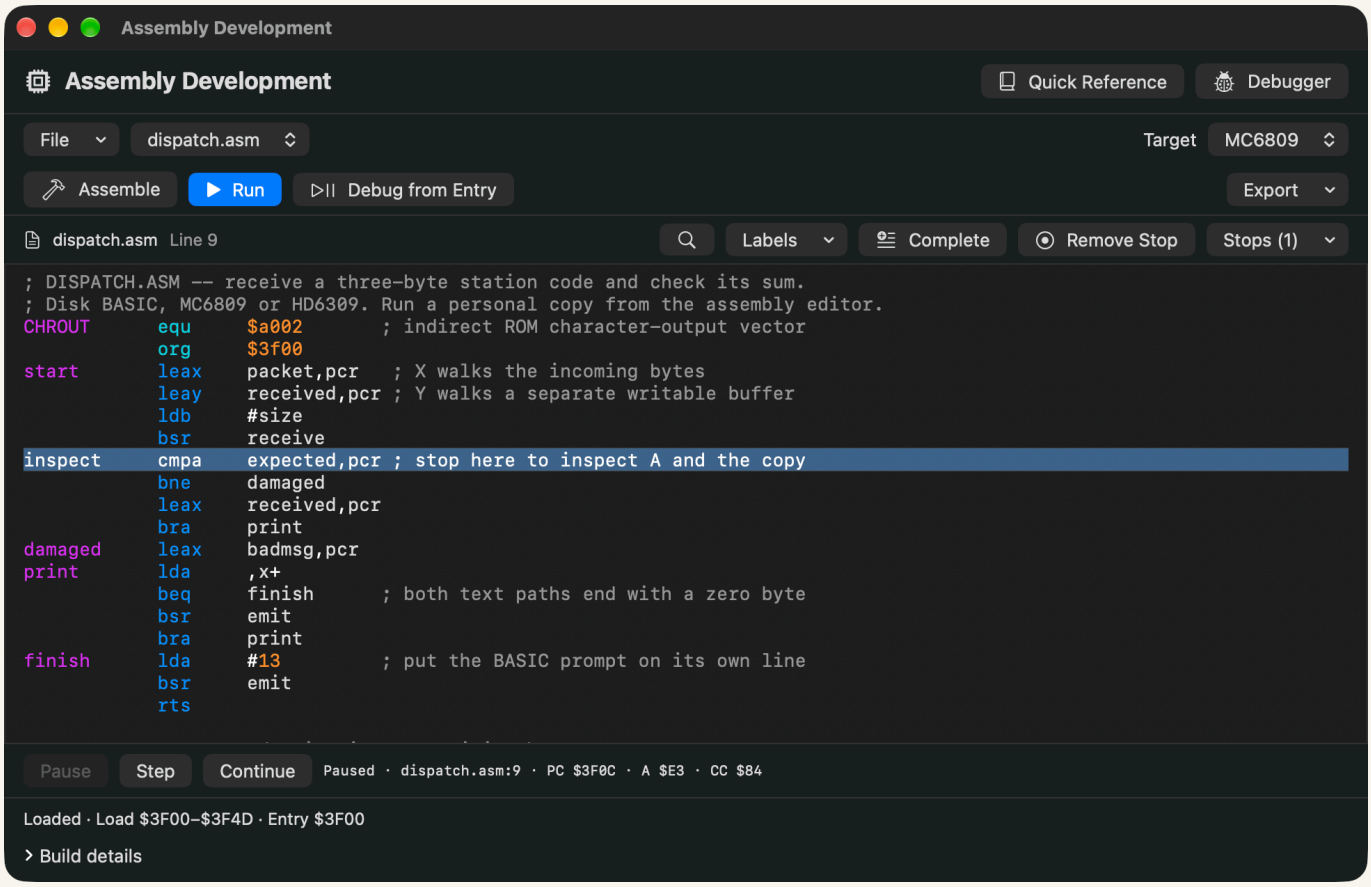


Figure 1: At the stop in dispatch.asm, the highlighted CMPA is next. The status bar already shows A=$E3.

Translate the listing into addresses

```
CHROUT      equ     $a002       ; indirect ROM character-output vector
            org     $3f00
```

```
start       leax    packet,pcr   ; X walks the incoming bytes
            leay    received,pcr ; Y walks a separate writable buffer
            ldb     #size
            bsr     receive
inspect     cmpa    expected,pcr ; stop here to inspect A and the copy
            bne     damaged
```

The **ORG** places the first instruction at $3F00. The **END** at the bottom supplies the entry label. X will point at the input and Y at the destination. **LDB #size** loads the count itself. Read **BSR receive** as "call receive and save where to come back." Execution is now back from that call. The instruction at the highlighted PC has yet to run.

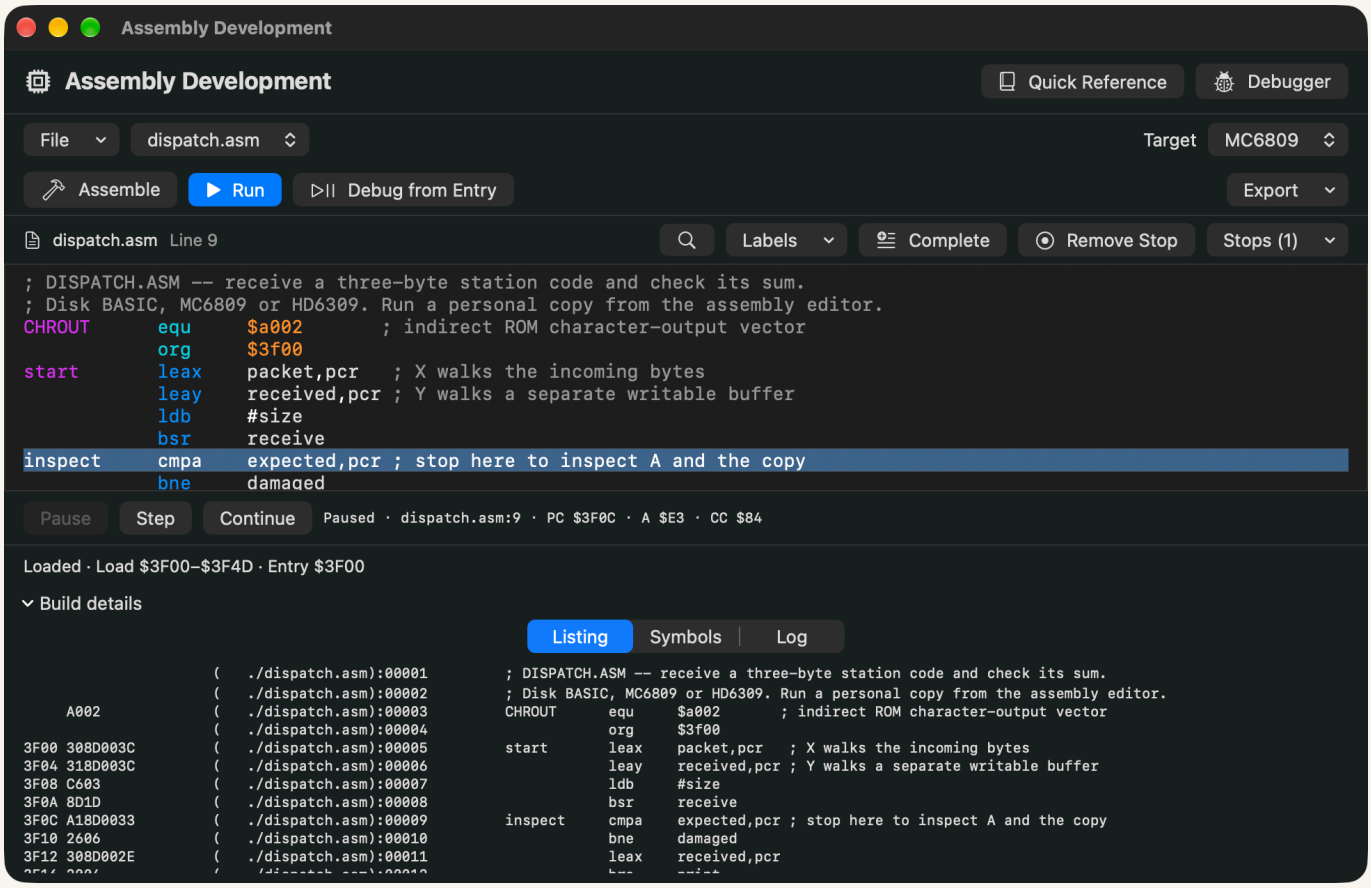


Figure 2: The assembly listing for this build shows the bytes of both LEA instructions, the call at $3F0A, and the comparison at $3F0C.

The first instruction occupies four bytes, **30 8D 00 3C**. LWASM used a signed 16-bit displacement in this build. The processor takes its position *after* the operand ($3F04) and adds $003C. That gives X the address $3F40. LEAX computes an address; it can do its job even before anyone looks at the character there. LEAY uses the same displacement but a different starting PC. Work it out from $3F08 before looking up **received**. Read your own listing after an edit: changing instruction length can move both labels.

Find the copy in RAM

In the workspace choose **Debugger**. Under **Inspector → Memory**, select **Mapped CPU**, enter **3F40** in **Address**, and choose **Go**. A row starts at the address printed on its left; the column adds the last hexadecimal digit. For this build, read across the row like this:

Addresses	Contents at the stop	What placed them here
$3F40–$3F42	4F 52 42	The assembler put the three character bytes of **packet** in the binary.
$3F43	E3	The author supplied the expected sum as data.
$3F44–$3F47	4F 52 42 00	The receive routine wrote the first three bytes. The fourth was initialized to zero and left alone.
$3F48 onward	43 48 45 43 4B 00	The other message, **CHECK**, followed by a terminator.

Open the registers. X=$3F43 and Y=$3F47 both point *one past* the last byte transferred. B has counted down to zero. In A, $E3 is a value calculated from the input. Compare it with the independent byte at $3F43. You have two pieces of evidence even though the guest has yet to print anything.

Return to the workspace and choose **Continue**. The guest prints ORB and returns to BASIC. Type **PRINT 2+3**, then Return. Seeing 5 tells you that BASIC is interpreting fresh input. Next you'll change exactly one input byte and catch the same decision.

Give the receiver a damaged signal

Chapter 2 of 8

A checksum buys you a second opinion about a message. In this receiver the sender and receiver agree that a healthy three-byte message adds up to $E3. You can change the message at the sending end while leaving that agreement in place.

Predict the next PC

Start with your portable receiver. At the bottom, **FCC "ORB"** lays down three character codes, with no length byte or terminating zero supplied by FCC. **size EQU *-packet **is the distance from the packet label to the current assembly position.** FCB $E3** supplies an expected value. Changing that value along with the message would spoil the experiment.

In the intact packet the partial sums are $4F, $A1 and $E3. Save a second copy and change only the quoted **ORB** to **ORD**. The code for D ($44) is two more than B ($42). Predict A after the third addition. Assemble this copy, set a stop on **inspect**, and Run. The destination now reads **4F 52 44 00**. The expected byte still reads **E3**.

With PC at inspect	Intact	Changed
A before CMPA	$E3	$E5
Z **after** stepping CMPA	1	0
PC after stepping BNE	$3F12, success path	$3F18, damaged path
Guest output after Continue	ORB	CHECK

The flag you see on arrival at **inspect** belongs to the earlier instructions. In particular, the last DECB made B zero on **both** runs. Step CMPA before interpreting Z as the result of the comparison. CMPA updates the condition codes as if it subtracted the expected byte; A keeps the received sum. BNE branches when Z is clear.

Ask what the sum can tell you

An eight-bit accumulator keeps the low eight bits of the sum. Carry out of the top bit is lost to the total unless a program explicitly uses it. Can this receiver distinguish the letters O, R, B in one order from the same letters in another? Work the sum on paper. Addition gives the same answer in both orders. A checksum like this can catch the edit you made. It cannot prove the message's order or authorship.

Make two correct comparisons with the same flags

Open **Develop → Assembly Quick Reference…** and choose the example **Compare signed and unsigned**. Open its independent example in the workspace. Use **Debug from Entry** on a disposable guest. Step LDA and CMPA. A holds $FF; the operand was $01. The subtraction used for the flags gives $FE. Z=0, N=1, V=0 and C=0. The comparison leaves A at $FF.

Read the next two branches before stepping them. **BLT** tests whether N differs from V after the subtraction. They differ here, so BLT goes to **signed**. The branch there, **BHI**, tests unsigned "higher": carry clear and zero clear. It goes to **both**. Branches use the flags from CMPA; taking the first branch preserved those flags for the second.

How can both be true? Read the same bits as a signed eight-bit number and $FF is −1, less than +1. Read them as an unsigned count and $FF is 255, greater than 1. The author chooses which question to ask by choosing the branch. The example holds at **both** for inspection. It deliberately loops there; use a fresh BASIC session for the next exercise. The printable receiver instead exits with RTS.

Give a subroutine a contract

Chapter 3 of 8

Imagine handing the receiver to someone who will call it from a second program. “Copies a message” leaves important questions open. Which register holds the length? Does it copy a zero? Who provides memory? Can the caller still find the next instruction after borrowing the stack?

Trace one trip around the loop

```
; Input: X source, Y destination, B positive byte count.
; Output: A = sum modulo 256. X and Y advance; B ends at zero.
receive     clra
next        adda    ,x          ; add this byte, leave X at it for the copy
            pshs    a           ; keep the running sum while moving the byte
            lda     ,x+
            sta     ,y+
            puls    a
            decb
            bne     next
            rts
```

On entry, X points at readable bytes, Y points at a separate writable destination, and B is a **positive** byte count. The caller owns both regions. It reserves one extra initialized zero for the later print routine. **receive** does its first copy before it tests B. Supplying zero here would make the first decrement wrap to $FF and the loop perform 256 transfers. That would run far past this destination. A general-purpose copy routine accepting an empty message would need an entry test. Keep the supplied three-byte count for this lab.

The first ADDA reads from X without moving X. PSHS saves the running sum because LDA needs to borrow A to carry a character. The plus in **,X+** takes the byte at the old address and then advances X by one. The store does the same for Y. PULS restores the saved sum. DECB sets the flag used by BNE. When the last iteration finishes, both pointers have advanced by the original count, B is zero, and A contains the checksum. CC has changed. Code that needs the original pointers or flags must save them or arrange to reconstruct them.

Find both kinds of saved byte

Start the portable receiver again with a stop on **BSR receive**. In the Inspector write down S. Follow the steps below in the assembly window; read memory through **Mapped CPU**. For a refresher, read the return address one byte at a time.

Stop before the instruction	Where to look	What was observed in Disk BASIC
BSR at $3F0A	S	$7F35
CLRA at the start of receive	New S and the byte above it	S=$7F33; **3F 0C** is the address of inspect.
LDA ,X+ immediately after the first PSHS A	Top three bytes starting at S	S=$7F32; **4F 3F 0C**. The one-byte sum sits below the two-byte return address.
DECB after PULS A	S and A	S=$7F33, A=$4F. The return address is again on top.
CMPA at inspect after the routine returns	S and PC	S=$7F35, PC=$3F0C. The caller can keep using its stack.

Use your own S if BASIC started with a different stack position. Set a second stop at **inspect** once you've followed the first pass; Continue lets the other passes finish. The saved sum and the saved PC use the same storage mechanism. The instruction that later consumes those bytes determines their meaning.

Choose how to address an operand

Form	What the CPU uses	Example to find
Immediate	A value embedded in the instruction	LDB #size puts 3 into B.
Direct	An eight-bit operand supplies the low byte; DP supplies the high byte	Inspect DP before using a direct-page location. A direct-page offset of $21 would name $0021 only when DP=$00.
Extended	A full 16-bit logical address	The pointer used for output is at $A002. This address names a ROM vector in this BASIC environment.
Indexed	A register supplies a base; an offset or update changes how it is used	ADDA ,X leaves X alone. LDA ,X+ advances X.
PC relative	A signed displacement is added to the PC after the operand	LEAX packet,PCR finds this program's data even if code and data move together.
Indirect	Memory holds the address to use next	The brackets in JSR [CHROUT] tell the CPU to read the two-byte destination address through the vector.

Cross the ROM boundary

```
emit        pshs    x           ; caller still needs its position after ROM I/O
            jsr     [CHROUT]
            puls    x,pc
```

Look at mapped addresses $A002 and $A003. Our Disk BASIC machine contained **A2 82**, so this call entered $A282. The source names the **vector** rather than baking that observed destination into the program. This service expects a character code in A. The caller still needs X to walk the text after the service returns; **emit** saves X and restores it. PULS X,PC also consumes the return address placed by the caller's BSR. A routine that branches into **emit** without that return address would give PULS the wrong stack layout.

ROM entry points are part of the environment. The supplied examples assume the indicated BASIC ROM is mapped and its services initialized. A program running in another OS needs its own service conventions. The linked receiver gives the copy routine a second caller without changing its contract.

Follow the same registers across an operating-system boundary in Beacon's OS-9 call contract.

Borrow a different byte at the same address

Chapter 4 of 8

A 6809 pointer fits in two bytes. It can name 65,536 locations, yet a 512 KB CoCo 3 has eight times that much RAM. The GIME, the chip that manages memory and video in the CoCo 3, chooses which physical memory answers each CPU address. You can catch it answering the same question two different ways.

Translate one address by hand

Open a fresh **Disk BASIC** setup with **512 KB**. In **Debug → Show Debugger**, choose **Inspector → MMU**. The eight rows divide the CPU's address space into windows of $2000 bytes (8 KB). Read the active task. Task 0 and task 1 are two sets of bank selections. CPU reads follow the active set when the MMU is enabled. **RAM backing** gives a physical RAM location. **Effective read** tells you whether the processor actually reads that RAM or a ROM overlay.

Find the row for CPU addresses $4000–$5FFF. On the practice machine the MMU is enabled, task 0 is active, and that row names bank $3A. The distance from $4000 to $4123 is $0123. Multiply the bank number by the bank size, then add the offset:

Selection	Start of physical bank	CPU $4123 reads at
Bank $3A	$3A × $2000 = $74000	$74123
Bank $01	$01 × $2000 = $02000	$02123

In **Memory**, switch between **Mapped CPU** and **Physical RAM**. Look up $4123 in the first and $74123 in the second. Record the values. Equal bytes are consistent with the map; two zeros alone would be poor evidence that the addresses are connected. Give the second bank a distinctive byte and the connection becomes visible.

Borrow block 2

Save bankwalk.asm and open that copy in Assembly Development. Use the fresh setup above. Before running, check that block 2 has effective RAM reads, the initial bank is $3A and the code starts at $3F00. If the mapping differs, stay with read-only inspection until you have the practice configuration. This example deliberately changes a machine register. Finish its restoration before running anything else.

```
start       pshs    cc,a            ; preserve incoming flags and A
            orcc    #$50            ; hold IRQ and FIRQ until restored
            lda     bank2
            anda    #$3f            ; 512 KB supplies six bank bits
            sta     oldbank,pcr
            lda     #1
            sta     bank2           ; window now sees physical $02123
            lda     window
            sta     oldbyte,pcr     ; keep the byte even on a fresh machine
            lda     #$a5
            sta     window
inspect     lda     oldbyte,pcr     ; pause BEFORE this instruction
            sta     window          ; put the borrowed byte back
            lda     oldbank,pcr
            sta     bank2           ; restore the original translation
```

The program saves its incoming flags and masks IRQ and FIRQ. It saves the six bank bits the 512 KB machine implements. Then it writes $01 to $FFA2, the task 0 register for block 2. From that instruction onward, an ordinary read or write at $4123 goes to the borrowed bank. The program remembers the byte already there, writes $A5, and leaves an instruction labeled **inspect** for you. The code and saved bytes live in block 1. The running BASIC stack is in block 3. Neither is moved by that register write.

Catch the mapping in the act

1. Choose **Debug from Entry**. Read the addresses of **inspect** and **restored** in **Build details**. In the supplied build they are $3F1E and $3F2C.
2. Open the machine **Debugger**. Add those two hexadecimal addresses in **Breakpoints**, then use the debugger's **Continue**. These are CPU address stops; assembly source debugging checks the memory mapping attached to its build. The address stops let you observe a deliberate mapping change.
3. At **inspect**, read the MMU row again. Task 0, block 2 now selects $01. In **Mapped CPU**, read $4123. In **Physical RAM**, read $02123 and $74123. The first two reads give $A5. The original bank still contains its old byte. Read **oldbank** and **oldbyte** from block 1 if you want to see how cleanup will work.
4. Continue to **restored**. Block 2 names $3A again, $4123 reads the original value, and $02123 holds its original byte. Remove both debugger breakpoints and Continue to BASIC. Enter `PRINT 2+3` to check the return. On the fresh practice machine both saved bytes from the two physical locations were zero. Restoration still matters when the borrowed location starts at zero.

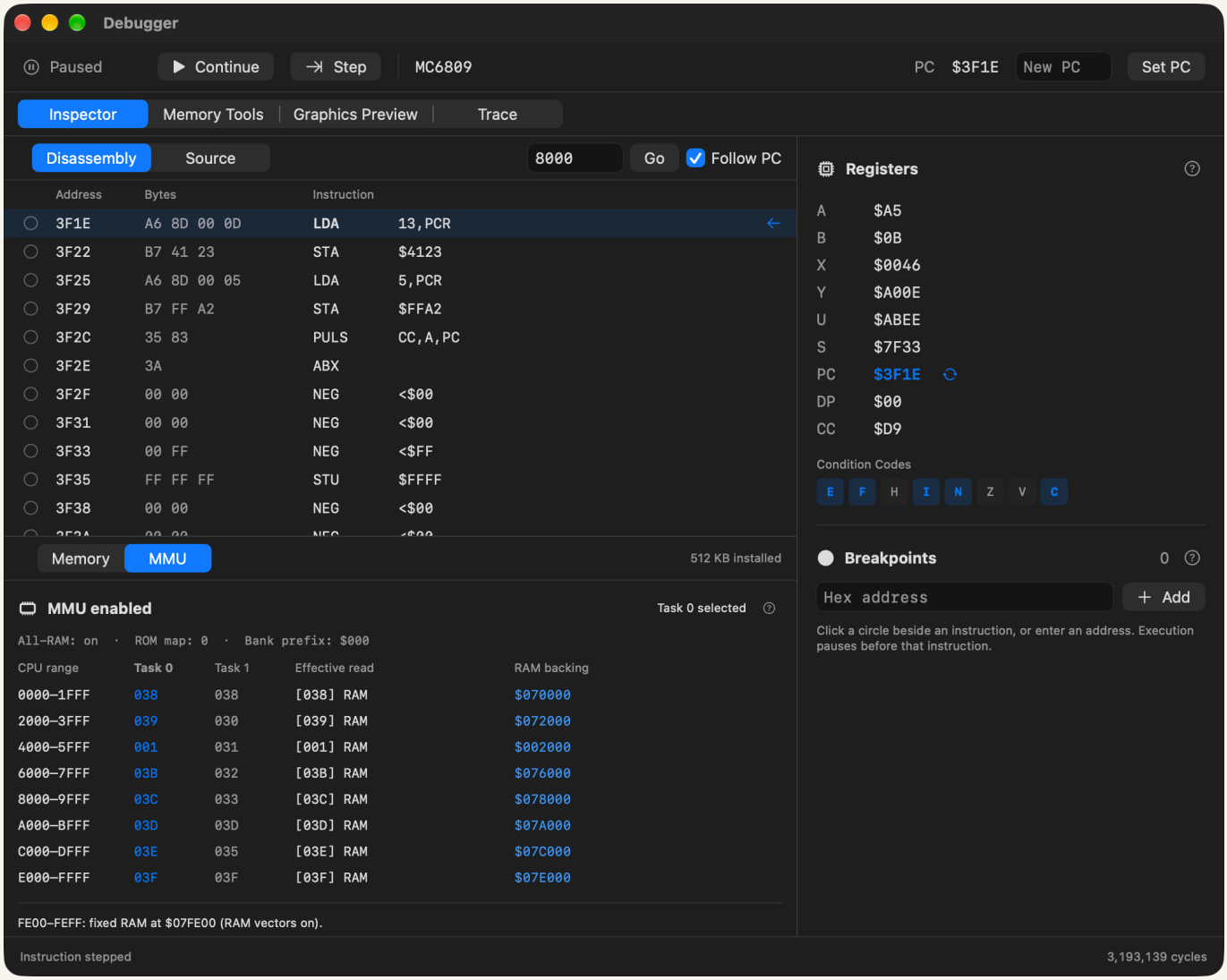


Figure 3: BANKWALK paused at $3F1E. Task 0 maps $4000–$5FFF to physical bank $001, beginning at $002000. Register A still holds the $A5 marker.

Read the exceptions on the map

An address can select a device as well as memory. A write to $FFA2 changes a bank assignment. A write to $FFB6 changes a video palette entry. A call through [$A002] fetches a two-byte destination from the BASIC character-output vector. Those three operations have very different consequences. For the top CPU block, consult the MMU view's ROM, vector and I/O notes before interpreting an address. The debugger can inspect physical RAM behind an overlay; a physical address cannot stand in for a read of a device register. Some device reads acknowledge an event. The debugger marks unavailable values where it cannot inspect safely.

The handbook's debugger guide shows the two memory views. For historical context on the CoCo 3 memory map, see Laurence A. Tepolt, *Assembly Language Programming for the CoCo 3* (TEPCO, 1987), figures 3-4 and 4-1.

Draw on the page you cannot see

Chapter 5 of 8

A display is a reader of memory. Let BASIC allocate two pictures, draw a mark in each, and leave the monitor on the first. The machine debugger can then show you a picture that nobody looking at the CoCo monitor can see. Afterward, use assembly to change a color without rewriting the letters on the screen.

Stop with both pictures in RAM

Choose fresh **Disk BASIC**, 512 KB, and **RGB** in **Settings → Video Signal**. Open PAGES.bas from Book 3 in BASIC Development, set a breakpoint at line **50**, and Run. The source starts in 32-column text, selects PMODE 1 and shows its first page. The filled square lands on the left. It then selects page 3 for drawing, clears it and draws a smaller square on the right. At the breakpoint the second page is complete. BASIC has yet to show it.

In **Debug → Show Debugger → Graphics Preview**, select **Current Video Page**. Next choose **At Address**, **Mapped CPU**, hexadecimal **1A00**, **Go**, **PMODE 1** and **Set 0**. Here are both preview buffers captured at the same stopped instruction:

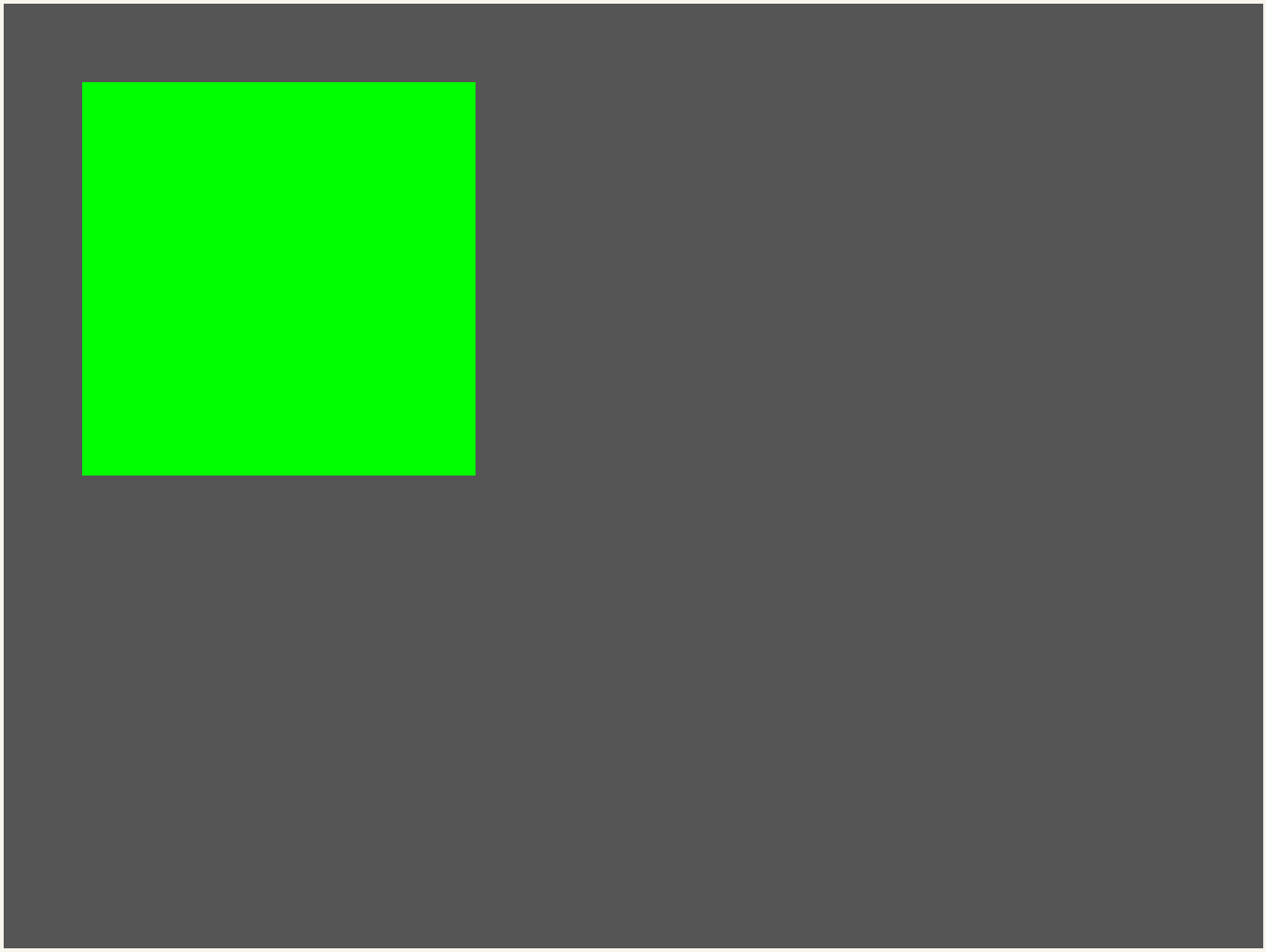


Figure 4: Current Video Page from the paused Disk BASIC guest. Its filled square is on the left.

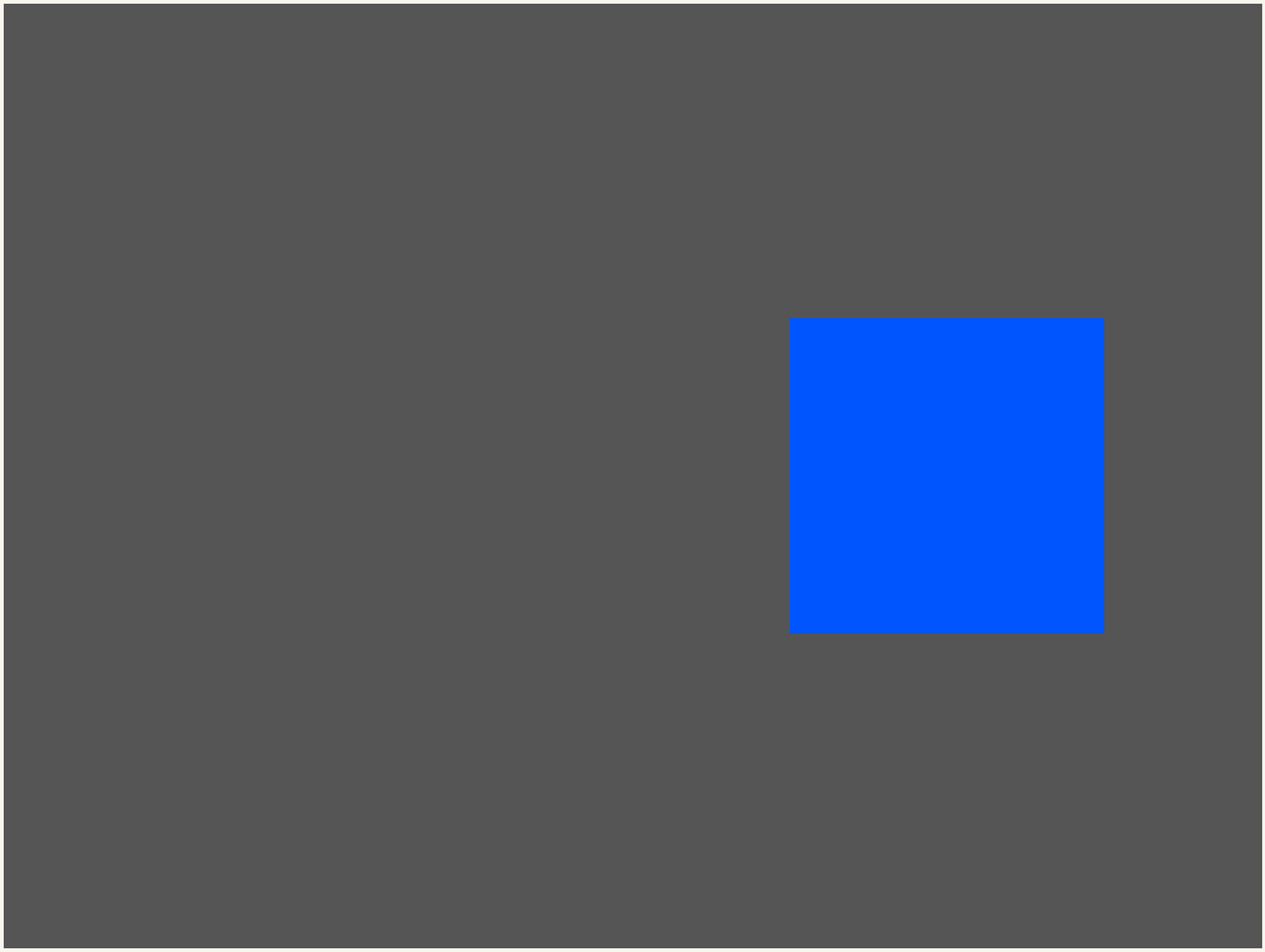


Figure 5: At Address, mapped CPU $1A00, from that same pause. Its filled square is on the right.

PMODE 1 stores 128 × 96 pixels with four two-bit pixels per byte. A row therefore occupies 32 bytes and a picture occupies 3,072 bytes ($0C00). On this setup the first picture corresponds to mapped $0E00 and the second to mapped $1A00. The Current Video Page inspector reported the first one in **physical** RAM at $70E00. Those numbers describe two different address spaces. In BASIC's page numbering this picture occupies two allocation pages; selecting page 3 leaves room for the whole first picture.

Work backward from pixel 81,32

On the hidden page select X **81**, Y **32** in Pixel Inspector. Predict the byte address before using **Show in Memory**. Row 32 begins $0400 bytes after the start ($20 rows × $20 bytes per row). Pixel 81 is in byte 20 ($14) of the row. Add them: $1A00 + $0400 + $14 = **$1E14**. Pixel X=80 occupies the highest pair of bits. X=81 occupies bits 5 and 4, with mask **$30**.

The guest byte is **$AA** (binary 10101010). Masking with $30 leaves $20. Shifting down four places gives the stored pixel code **2**. That code selects a color in the active palette set. BASIC's drawing coordinates and pen numbers sit a level above these packed bits. Book 3's graphics lesson gives the original drawing and a native-window view of the inspector. To prove that you have isolated a single pixel, erase one beacon in hidden RAM and restore it.

Remove the line-50 breakpoint and Continue. Give the CoCo display keyboard focus. Tap and release a letter. The picture you found in RAM is now on the monitor. Another tap brings back the first square, and a third returns

to 40-column text with **BOTH PAGES ARE HOME**. Swapping which page is visible saved redrawing the first square.

Change the paint used by a text cell

Open Earth or Mars in assembly. This time use its CoCo 3 **40-column text** screen. Locate **greet**, set a source stop on `STB BG6`, and Run. Give the CoCo window focus and press **2**. At the stop the source has chosen the Mars string. In the supplied build PC is $3F22, X is $3F8E and B holds $25. Inspect the palette byte at $FFB6 in mapped memory. Its six color bits were $09 on the fresh setup.

Step one instruction. PC advances to $3F25 and the six color bits now read $25. On a repeat with **1**, B holds $0A at the same instruction and the write changes those bits to $0A. This is a change to GIME palette slot 6. The following instructions set BASIC's text attribute so its character-output routine draws the greeting with that background slot. The output routine still receives ordinary character codes. Changing the palette entry can alter how an existing pixel or cell using that slot looks without changing the byte containing its pixel code.

Continue. Read the colored greeting and the BASIC prompt beneath it. The starter clears the text attribute before returning. Its palette write persists; the return restores the text-writing convention, not every palette register. The two guest captures show the rendered choices. For CoCo 3 native high-resolution graphics rather than PMODE or text, Star Chart draws a CoCo 3 HSCREEN with separately defined palette slots.

Move the packet with the HD6309

Chapter 6 of 8

Try the same transfer with a different agreement between the program and the CPU. The portable loop uses B as a counter, A as a sum, and several instructions per byte. The Hitachi processor has a block-transfer instruction and a separate count register. You can see exactly which work it takes over.

Choose both processors deliberately

Save your guest work. In **Machine Settings**, choose the **Hitachi HD6309** processor and apply the restart. Start at the Disk BASIC prompt. In **Assembly Development**, open a copy of dispatch6309.asm and set the assembler's **Target** to HD6309. The Machine setting chooses what executes opcodes. Target chooses what the assembler accepts. As a negative check, choose the MC6809 target and **Assemble**: LDW and TFM are rejected. Return the target to HD6309 before continuing.

```
start       leax    packet,pcr
            leay    received,pcr
            ldw     #size       ; W counts the bytes left to transfer
copy        tfm     x+,y+       ; move from X to Y and increment both
inspect     leax    received,pcr ; stop here to inspect W, X, Y and RAM
next        lda     ,x+
```

W is a sixteen-bit register made from E (high byte) and F (low byte). D is similarly made from A and B. On the 6309, Q is the 32-bit combination D:W. The E **register** and the E **flag in CC** have different jobs. Watch W through E and F if the inspector shows the halves separately.

Cross the transfer in one debugger step

Set a stop on **copy**, choose Run, and look up **packet** and **received** in Symbols. At TFM, X points at the first input byte, Y at the first destination byte, and W is 3. Predict the pointer values after the copy. Step once, then inspect the four bytes at received in **Mapped CPU**.

Value	At copy	At inspect
PC in this build	$3F0C	$3F0F
X	$3F28	$3F2B
Y	$3F2B	$3F2E
W	$0003	$0000
Four destination bytes	00 00 00 00	4F 52 42 00

TFM X+,Y+ copies a byte, advances both pointers and counts down W. In VCC Plus the debugger presents this transfer as a whole instruction. The stop after it is where you compare the buffers. Continue prints ORB and returns. W supplied the count, and the fourth zero came from the original buffer declaration. This version never computes a checksum. Comparing its total run time with the first receiver would mix two different jobs.

Observe native mode, then give BASIC its mode back

The HD6309 can execute the preceding instructions in its startup compatibility mode. Native mode is a *separate* processor setting with effects including interrupt frames and instruction timing. To observe the distinction, use a freshly restarted **HD6309** Disk BASIC guest. Save and open mode.asm. Set a stop on **native** and choose Run. The exercise assumes the reset value MD=$00. It saves CC and masks IRQ and FIRQ during its short mode-changing section. Keep it as supplied and let it finish.

```
start           pshs    cc              ; remember BASIC's interrupt masks
                orcc    #$50            ; mask IRQ and FIRQ for the short experiment
                ldmd    #1              ; select 6309 native mode
native          ldw     #$1234          ; inspect MD, then step and read E and F
                ldmd    #0              ; restore reset mode before calling BASIC
restored        puls    cc              ; restore the caller's condition codes
                rts
```

At **native** the instruction that selects mode 1 has executed. In the Debugger's HD6309 registers inspect MD. Step LDW: E becomes $12 and F becomes $34. Step LDMD #0. At **restored**, MD reads $00 while the data in W still represents $1234. Continue restores the saved condition codes and returns to BASIC. Enter **PRINT 2+3** to check that the guest is listening again.

Changing mode around an unknown interrupt handler would give that code a different environment. In particular, a full native-mode interrupt frame includes E and F. The interrupt experiment starts from a known mode and lets you inspect an actual frame. If you interrupt the mode experiment and lose your place, restart the disposable guest before continuing the book.

Read the return ticket an interrupt leaves

Chapter 7 of 8

A subroutine call saves the address where its caller can resume. An interrupt can arrive while several registers hold half-finished work. How much more does the CPU have to remember? Trigger the event yourself with **SWI** and compare the stack before and after it. You control when the experiment starts, and you can finish it without waiting for an external device.

Follow the vector to a RAM stub

Use a fresh Disk BASIC practice session, with the MC6809 or an HD6309 in its normal emulation mode. Save softirq.asm. Before running it, open the machine debugger and inspect these mapped bytes:

Address	Bytes on the practice machine	What uses them
$FFFA	FE FA	The CPU reads this 16-bit SWI vector and sets PC to $FEFA.
$FEFA	16 02 09	Opcode $16 is LBRA. The displacement is relative to $FEFD, the PC after the instruction. Adding $0209 wraps to $0106.
$0106	00 00 00	Disk BASIC's three-byte RAM stub on this machine. The example saves all three bytes before installing its handler.

If your vector chain differs, leave it alone and use the supplied observations or return to the practice profile. The CPU vector gives an address. The relay at that address is executable code. The stub at the relay destination will become another instruction, an extended JMP to our own handler. These are three distinct things you can identify in memory.

The program saves its caller's CC, D and X, masks IRQ and FIRQ, and installs the temporary JMP. The masks hold off those hardware interrupt classes during the experiment; executing SWI still triggers a software interrupt. Keep the guest in this program through cleanup. If you abandon the experiment before restoration, cold-restart the disposable profile before using it for anything else.

Catch the frame before the handler touches it

Set source breakpoints at **trigger**, **handler** and **returned**, then Run. Read the addresses from the listing after any edit. At **trigger** the supplied 6809 build has PC=$3F26, A=$4F, B=$52 and S=$7F30. The values in A and B spell the first two characters of our station code. Continue to **handler** and look at S **before** executing INC. It now reads $7F24, twelve bytes below its value at SWI.

Switch the machine debugger to **Mapped CPU** memory and look up your handler's S. If you use a different BASIC stack, look at the address in the S register rather than copying $7F24. In increasing address order the test guest showed:

At S + offset	Saved register	Bytes in this run
0	CC	D0
1, 2, 3	A, B, DP	4F 52 00
4–5	X	3F 37
6–7	Y	A0 0E

At S + offset	Saved register	Bytes in this run
8–9	U	AB EE
10–11	PC	3F 27

Multibyte registers have their high byte first. Locate $4F and $52 as a check on your orientation. The final two bytes deserve a closer look. SWI at $3F26 is one byte long; **$3F27** is the instruction immediately after it. The processor stacked a return PC that will let the program carry on. E, bit 7 of the stacked CC, is set. For the 6809 this indicates a full frame.

```
            lda     #$4f            ; known bytes to locate in the stack frame
            ldb     #$52
trigger     swi                     ; CPU stacks state and follows the SWI vector
returned    lda     oldop,pcr        ; breakpoint to compare S after RTI
            sta     hook
            ldx     oldaddr,pcr
            stx     hook+1
restored    puls    cc,d,x,pc        ; restore flags, D and X, return to BASIC

handler     inc     visits,pcr      ; memory survives the return
            rti                     ; reload the stacked context, including PC
```

Let RTI rebuild the context

Continue to **returned**. The handler increments a byte named **visits** and executes RTI. Inspect **visits** using its listing address ($3F3F in the supplied build). It is now 1. Compare registers with the trigger stop: A=$4F, B=$52 and S=$7F30 are back, and PC is the saved $3F27. The memory side effect survives the return. The stack frame has been consumed; old bytes might still be visible below S until some later call reuses that memory.

Set a final source stop at **restored**, then Continue. Read $0106–$0108 again. The original three bytes are back. Continue once more and enter `PRINT 2+3` at the BASIC prompt. An RTS here in place of the handler's RTI would take the top two bytes as an address and leave the rest of the saved context on the stack. You can explain that failure from the table without trying it.

Apply the observation to other interrupts

IRQ and **FIRQ** are signals a device can assert. A real interrupt handler also has to determine which device needs service and acknowledge it according to that device's rules. SWI gives us a repeatable stack experiment without those timing or device interactions.

For the MC6809, IRQ normally saves the full register frame. FIRQ normally saves just CC and PC, with the stacked E bit clear. Code in a fast handler has to preserve other registers it uses. RTI reads the stacked state to determine what to pull. The HD6309 can run in emulation or native mode; full-frame stacking in native mode also includes E and F, and the 6309 provides a mode control affecting FIRQ stacking. Recheck the mode before applying any fixed frame diagram. The twelve bytes shown here were measured on both supported processors in the startup configuration. For full-frame layouts and mode controls see Darren Atkinson, *Motorola 6809 and Hitachi 6309 Programmer's Reference* (2009), parts I–II.

Deliver the receiver

Chapter 8 of 8

Imagine handing your program to someone who has only a CoCo and Disk BASIC. They cannot open your source files and click **Run**. They can load a binary from a floppy. In this chapter you build a single executable from two source files, catch the call where they meet, and give your reader two ways to run the finished disk.

Open the whole project

Save the Dispatch project, expand the ZIP in a working folder, and open **Develop → Assembly Development...**. Choose **File → Open Project Folder...** inside that window and select the folder containing these four files. Start with the **Disk BASIC** profile at its prompt and an **MC6809** assembler target. Use a personal copy of any mounted media.

File	Its job
`main.asm`	Supplies the entry point, packet, expected checksum, destination buffer and screen output.
`receive.asm`	Copies bytes and computes their checksum. The caller supplies X, Y and B.
`vccplus-assembly.json`	Identifies the entry source, additional module and linker script for VCC Plus.
`layout.link`	Places the combined code at hexadecimal 3F00 and selects the symbol `start` as the entry.

The two assembly files each name a section called `code`. Read the first few lines of each. The display module **imports** `receive`, promising to find its definition in another object. The routine **exports** that name, allowing the linker to resolve it. The files can agree on an address and still disagree about what the routine expects. Find the instructions in the caller that satisfy all three inputs in the comment above `receive`.

Read the address that the linker chose

Choose **Assemble**. Expand **Build details** below the source. **Log** shows two assembler invocations followed by the linker. **Listing** describes the assembly of each object. **Symbols** shows where the linker actually placed those sections and symbols. A zero offset at the start of the second object's section can therefore be a perfectly good listing value.

Symbol or instruction	Initial build	What to look for
`start`	$3F00	First byte of the entry module. The linker reports its section length as $0040.
`jsr receive`	$3F0A	Bytes BD 3F 40. The final two bytes contain the absolute address supplied by the linker.
`inspect`	$3F0D	Next instruction after the call. JSR puts this return address on the stack.
`receive`	$3F40	First instruction supplied by the second object. Its section takes $000F bytes.

Select `receive.asm` in the project and place the caret on `receive clra`. Choose **Set Stop**. Set another stop on `inspect cmpa expected,pcr` in `main.asm`. **Run** and confirm the load if prompted. The first stop is in the other module, before CLRA executes. B holds 3. If you examine the two bytes at S in the debugger's mapped memory, they are 3F 0D. The subroutine has a route back into its caller even though the two files were assembled separately.

Continue to `inspect`. A holds $E3 and the four bytes at `received` ($3F36 in this build) are 4F 52 42 00. Continue again. The CoCo prints `ORB` and returns to BASIC. For a deliberate failure, edit the quoted `ORB` in `packet` to `ORD`, leaving `expected` alone. Run the new build. A is now $E5 at the comparison and the CoCo prints `CHECK`. Restore the letter B before you make the disk for your reader.

Move the code and keep the source stop

Try this on a separate copy of the entire project. In `layout.link` change the first line from `section code load 3f00` to `section code load 4100`. Leave the second line, `entry start`, alone. Save, then **Assemble**. Before looking at **Symbols**, predict the second module's new address from the length of the first module.

The source stops are still the CLRA in `receive.asm` and the comparison in `main.asm`. Set them if you opened a fresh copy. On **Run**, the same CLRA now stops at $4140. The stack contains 41 0D. Continue to $410D and check A before finishing. The linker changed the absolute target of JSR to $4140. The packet and its PC-relative load moved together, so their displacement stayed the same. Neither a source stop nor a symbolic operand required you to type an address into the instructions.

The two tested placements are for this short program in the lab's fresh Disk BASIC session. Choosing a new load address also means finding space the guest can spare. A writable address can still contain a BASIC program or another routine you need.

Hand someone a disk

Return to the unmodified $3F00 project. The **Export** menu offers the DECB binary by itself and a floppy image containing that binary. A loose binary is useful for emulator quickload. Make two new images now to test what someone can do at a guest prompt.

1. Choose **Export → Export Disk Image…**. Enter `DISPATCH` as the **Disk BASIC filename**, leave **Install DOS bootloader** clear, choose **Choose Location…**, and save as `Dispatch-plain.dsk`. The eight-letter guest name and the Mac filename serve different purposes.
2. Open the floppy controls. Preserve anything you need from the current FD0 disk and mount your new image in FD0. At the fresh **Disk BASIC** prompt, type `DIR` and press Return. Look for `DISPATCH BIN`. Enter `LOADM"DISPATCH":EXEC`. LOADM puts the records into their assigned addresses and records the entry point. EXEC transfers control there. The greeting for an intact packet is `ORB`.
3. Export again with the same guest name. This time select **Install DOS bootloader** and save the new image as `Dispatch-boot.dsk`. Mount this image in FD0. Type `DOS` on its own. Watch the line underneath it as the loader supplies the full Disk BASIC command. You can still enter `LOADM"DISPATCH":EXEC` yourself on this disk.

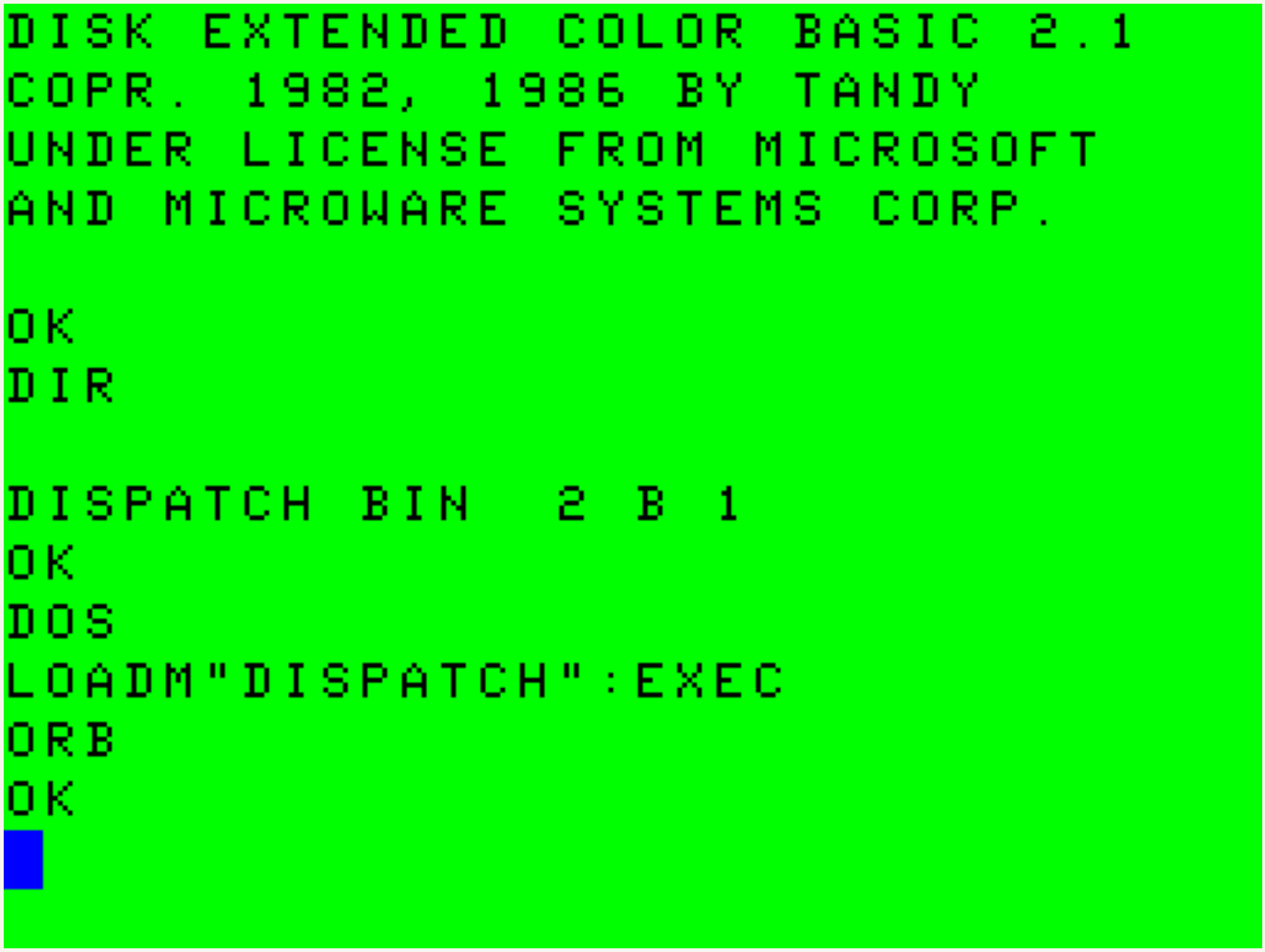


Figure 6: Disk BASIC lists DISPATCH BIN. After DOS, the bootloader supplies LOADM"DISPATCH":EXEC; the receiver prints ORB and BASIC displays OK.

The drive starts the same machine-language program either way. The source stops from the Assembly workspace belong to its direct debugging session. For the disk test, judge success at the CoCo screen. After the final `OK`, enter `PRINT 2+3`. Seeing 5 proves that the program returned with BASIC able to accept and execute another command.

A handoff you can repeat

Give a developer the four project files together. Give someone who wants to run the program the DSK, the required **Disk BASIC** profile, **FD0**, the name `DISPATCH`, and the exact launch command. Identify whether that particular disk has the optional bootloader. Include the expected output and how control returns.

If the project assembles but does not stop where expected, compare its current **Symbols** map with the source file and instruction you selected. If `DIR` lists the wrong contents, check which image is in FD0 before changing any source. If the disk outputs `CHECK`, it has already loaded and started the receiver. Look at the packet and checksum of the project you exported. Those observations separate a build problem, a mounting problem and a working program reporting damaged input.

Instruction directory for the 6809 and 6309

Appendix A

You have stopped on `BNE`. Does it test the accumulator? It reads one bit in CC. Look up BNE, then find the instruction that last wrote Z. In the receiver's counting loop, that writer is DECB. At the packet check it is CMPA. Identical branch instructions answer different questions because they inherit different evidence.

Pick an operation

What do you want the processor to do?	Go to
Get a value, store a value, or calculate where it lives	Move a value or compute an address
Add, subtract, multiply, divide, or extend a signed value	Calculate with integers
Test, clear, set, invert, or combine bits	Test and change bits
Move bits and use carry between operations	Shift and rotate
Compare numbers or select the next instruction	Compare and choose a path
Call a routine, protect registers, or return to a caller	Call, save and return
Interpret an interrupt, a processor mode or a wait	Interrupts and processor control
Copy a sequence of bytes with the HD6309	Transfer a block

Each family begins with an alphabetical list. Alternative spellings lead to the same entry. Search this book from its contents page to find a particular mnemonic. In the Assembly editor, put the caret on an instruction and choose **Look Up in Assembly Quick Reference** for the corresponding native topic.

Read a card from the listing

An instruction card identifies the processor, operands, operation, flags and a printed source. The operand line gives patterns. Pick one pattern and supply your register, constant, expression or label. A blank operand field in the listing is normal for an inherent instruction such as `RTS`. The register and addressing card explains the punctuation.

MC6809 and HD6309 means the hardware operation exists on both. **HD6309 only** means the guest needs the Hitachi processor for that hardware instruction. These entries can execute in its startup emulation mode as well as native mode. Selecting the 6309 assembler target admits the encodings. Selecting the 6309 processor supplies the hardware. The program's mode register controls execution mode and affects interrupt behavior and timing. See HD6309 modes before changing it.

Read **Opcode bytes** from left to right. For an indexed LDA the opcode is `A6`. A postbyte follows to say how to use the index register. For a long conditional branch, `10 26` selects LBNE, and the next two bytes hold the displacement. The directory gives just the opcode for each encoding. The listing in the Assembly workspace shows the complete instruction, operands included.

Flags describe the documented processor. When a flag is undefined, choose another test. If you need an earlier comparison, branch before issuing an instruction that rewrites its evidence. The signed comparison experiment shows how to read N and V together. The DIVQ entry flags differences in VCC Plus's overflow and carry results. For a trace of TFM, debugger Step completes a transfer, whereas Continue can encounter an instruction breakpoint again between bytes.

There are 215 hardware instruction entries and 234 hardware spellings in this edition. An alias can have its own operand convention. For example, BHS is another spelling of BCC; the TFM aliases take bare register names and encode their direction in the mnemonic. For multiple-instruction expansions, data declarations, and link directives, use the LWASM index.

Move a value or compute an address

In the lab, follow the bytes of the receiver.

`ABX` · `EXG` · `LDA` · `LDB` · `LDD` · `LDE` · `LDF` · `LDQ` · `LDS` · `LDU` · `LDW` · `LDX` · `LDY` · `LEAS` · `LEAU` · `LEAX` · `LEAY` · `STA` · `STB` · `STD` · `STE` · `STF` · `STQ` · `STS` · `STU` · `STW` · `STX` · `STY` · `TFR`

ABX

MC6809 and HD6309

```
No operand
```

Opcode bytes. `3A`.
Add the unsigned byte in B to X.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 4 · Motorola (1981), Appendix A, ABX · Astle, LWTools 4.25, ch. 3.

EXG

MC6809 and HD6309

```
source register,destination register
```

Opcode bytes. `1E`.
Exchange the contents of the two registers.
Flags. Writing CC replaces its flags.
Use equal-width registers on the MC6809. The HD6309 has additional registers and defined mixed-width behavior. Writing PC transfers control.
Reading: Atkinson (2009), pp. 65–66 · Motorola (1981), Appendix A, EXG · Astle, LWTools 4.25, ch. 3.

LDA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `96` · indexed `A6` · extended `B6` · immediate `86`.
Copy the operand into A. Use # for a literal or read one byte from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
LDA #$20 places $20 in A. LDA $0400 reads the byte at $0400. LDA ,X+ reads at X, then advances X by one. The remaining CC bits keep their values. Immediate LDA takes 2 cycles. Extended LDA takes 5 on MC6809 or HD6309 emulation mode and 4 in HD6309 native mode. Indexed timing depends on the postbyte.
Reading: Atkinson (2009), pp. 90 · Motorola (1981), Appendix A, LDA · Astle, LWTools 4.25, ch. 3.

LDB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D6` · indexed `E6` · extended `F6` · immediate `C6`.
Copy the operand into B. Use # for a literal or read one byte from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 90 · Motorola (1981), Appendix A, LDB · Astle, LWTools 4.25, ch. 3.

LDD

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `DC` · indexed `EC` · extended `FC` · immediate `CC`.
Copy the operand into D. Use # for a literal or read two bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 91 · Motorola (1981), Appendix A, LDD · Astle, LWTools 4.25, ch. 3.

LDE

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 96` · indexed `11 A6` · extended `11 B6` · immediate `11 86`.
Copy the operand into E. Use # for a literal or read one byte from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 90 · Astle, LWTools 4.25, ch. 3.

LDF

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 D6` · indexed `11 E6` · extended `11 F6` · immediate `11 C6`.
Copy the operand into F. Use # for a literal or read one byte from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 90 · Astle, LWTools 4.25, ch. 3.

LDQ

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 DC` · indexed `10 EC` · extended `10 FC` · immediate `CD`.
Copy the operand into Q. Use # for a literal or read four bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 94 · Astle, LWTools 4.25, ch. 3.

LDS

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 DE` · indexed `10 EE` · extended `10 FE` · immediate `10 CE`.

Copy the operand into S. Use # for a literal or read two bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 91 · Motorola (1981), Appendix A, LDS · Astle, LWTools 4.25, ch. 3.

LDU

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `DE` · indexed `EE` · extended `FE` · immediate `CE`.
Copy the operand into U. Use # for a literal or read two bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 91 · Motorola (1981), Appendix A, LDU · Astle, LWTools 4.25, ch. 3.

LDW

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 96` · indexed `10 A6` · extended `10 B6` · immediate `10 86`.
Copy the operand into W. Use # for a literal or read two bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
LDW #$1234 encodes as 10 86 12 34. E becomes $12 and F becomes $34 in either HD6309 execution mode.
Reading: Atkinson (2009), pp. 91 · Astle, LWTools 4.25, ch. 3.

LDX

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `9E` · indexed `AE` · extended `BE` · immediate `8E`.
Copy the operand into X. Use # for a literal or read two bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 91 · Motorola (1981), Appendix A, LDX · Astle, LWTools 4.25, ch. 3.

LDY

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 9E` · indexed `10 AE` · extended `10 BE` · immediate `10 8E`.
Copy the operand into Y. Use # for a literal or read two bytes from the addressed memory.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 91 · Motorola (1981), Appendix A, LDY · Astle, LWTools 4.25, ch. 3.

LEAS

MC6809 and HD6309

```
offset,index
```

Opcode bytes. 32.

Calculate an indexed address and place it in S. Indirect addressing reads a pointer before producing the address.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 95 · Motorola (1981), Appendix A, LEAS · Astle, LWTools 4.25, ch. 3.

LEAU

MC6809 and HD6309

```
offset,index
```

Opcode bytes. `33`.
Calculate an indexed address and place it in U. Indirect addressing reads a pointer before producing the address.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 95 · Motorola (1981), Appendix A, LEAU · Astle, LWTools 4.25, ch. 3.

LEAX

MC6809 and HD6309

```
offset,index
```

Opcode bytes. `30`.
Calculate an indexed address and place it in X. Indirect addressing reads a pointer before producing the address.
Flags. Only Z changes, to indicate a zero address.
Reading: Atkinson (2009), pp. 95 · Motorola (1981), Appendix A, LEAX · Astle, LWTools 4.25, ch. 3.

LEAY

MC6809 and HD6309

```
offset,index
```

Opcode bytes. `31`.
Calculate an indexed address and place it in Y. Indirect addressing reads a pointer before producing the address.
Flags. Only Z changes, to indicate a zero address.
Reading: Atkinson (2009), pp. 95 · Motorola (1981), Appendix A, LEAY · Astle, LWTools 4.25, ch. 3.

STA

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `97` · indexed `A7` · extended `B7`.
Copy A to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 127 · Motorola (1981), Appendix A, STA · Astle, LWTools 4.25, ch. 3.

STB

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D7` · indexed `E7` · extended `F7`.
Copy B to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 127 · Motorola (1981), Appendix A, STB · Astle, LWTools 4.25, ch. 3.

STD

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `DD` · indexed `ED` · extended `FD`.
Copy D to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 128 · Motorola (1981), Appendix A, STD · Astle, LWTools 4.25, ch. 3.

STE

HD6309 only

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 97` · indexed `11 A7` · extended `11 B7`.
Copy E to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 127 · Astle, LWTools 4.25, ch. 3.

STF

HD6309 only

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 D7` · indexed `11 E7` · extended `11 F7`.
Copy F to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 127 · Astle, LWTools 4.25, ch. 3.

STQ

HD6309 only

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 DD` · indexed `10 ED` · extended `10 FD`.
Copy Q to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 130 · Astle, LWTools 4.25, ch. 3.

STS

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 DF` · indexed `10 EF` · extended `10 FF`.
Copy S to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 128 · Motorola (1981), Appendix A, STS · Astle, LWTools 4.25, ch. 3.

STU

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `DF` · indexed `EF` · extended `FF`.
Copy U to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 128 · Motorola (1981), Appendix A, STU · Astle, LWTools 4.25, ch. 3.

STW

HD6309 only

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 97` · indexed `10 A7` · extended `10 B7`.
Copy W to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 128 · Astle, LWTools 4.25, ch. 3.

STX

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `9F` · indexed `AF` · extended `BF`.
Copy X to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 128 · Motorola (1981), Appendix A, STX · Astle, LWTools 4.25, ch. 3.

STY

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 9F` · indexed `10 AF` · extended `10 BF`.
Copy Y to the destination address. Multi-byte stores write the high byte first.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 128 · Motorola (1981), Appendix A, STY · Astle, LWTools 4.25, ch. 3.

TFR

MC6809 and HD6309

```
source register,destination register
```

Opcode bytes. `1F`.
Copy the source register to the destination.
Flags. Writing CC replaces its flags.
Use equal-width registers on the MC6809. The HD6309 has additional registers and defined mixed-width behavior. Writing PC transfers control.
Reading: Atkinson (2009), pp. 137–138 · Motorola (1981), Appendix A, TFR · Astle, LWTools 4.25, ch. 3.

Calculate with integers

In the lab, observe the checksum and its flags.
`ADCA` · `ADCB` · `ADCD` · `ADCR` · `ADDA` · `ADDB` · `ADDD` · `ADDE` · `ADDF` · `ADDR` · `ADDW` · `DAA` · `DEC` · `DECA` · `DECB` · `DECD` · `DECE` · `DECF` · `DECW` · `DIVD` · `DIVQ` · `INC` · `INCA` · `INCB` · `INCD` · `INCE` · `INCF` · `INCW` · `MUL` · `MULD` · `NEG` · `NEGA` · `NEGB` · `NEGD` · `SBCA` · `SBCB` · `SBCD` · `SBCR` · `SEX` · `SEXW` · `SUBA` · `SUBB` · `SUBD` · `SUBE` · `SUBF` · `SUBR` · `SUBW`

ADCA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `99` · indexed `A9` · extended `B9` · immediate `89`.
Add the operand and incoming carry to A. Keep the low 8 bits in A.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H records carry from bit 3.
Reading: Atkinson (2009), pp. 5 · Motorola (1981), Appendix A, ADCA · Astle, LWTools 4.25, ch. 3.

ADCB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D9` · indexed `E9` · extended `F9` · immediate `C9`.
Add the operand and incoming carry to B. Keep the low 8 bits in B.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H records carry from bit 3.
Reading: Atkinson (2009), pp. 5 · Motorola (1981), Appendix A, ADCB · Astle, LWTools 4.25, ch. 3.

ADCD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 99` · indexed `10 A9` · extended `10 B9` · immediate `10 89`.
Add the operand and incoming carry to D. Keep the low 16 bits in D.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H is preserved.
Reading: Atkinson (2009), pp. 6 · Astle, LWTools 4.25, ch. 3.

ADCR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 31`.
Add source and carry to destination.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 7 · Astle, LWTools 4.25, ch. 3.

ADDA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `9B` · indexed `AB` · extended `BB` · immediate `8B`.
Add the operand to A. Keep the low 8 bits in A.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H records carry from bit 3.
Reading: Atkinson (2009), pp. 8 · Motorola (1981), Appendix A, ADDA · Astle, LWTools 4.25, ch. 3.

ADDB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `DB` · indexed `EB` · extended `FB` · immediate `CB`.
Add the operand to B. Keep the low 8 bits in B.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H records carry from bit 3.
Reading: Atkinson (2009), pp. 8 · Motorola (1981), Appendix A, ADDB · Astle, LWTools 4.25, ch. 3.

ADDD

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D3` · indexed `E3` · extended `F3` · immediate `C3`.
Add the operand to D. Keep the low 16 bits in D.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H is preserved.
Reading: Atkinson (2009), pp. 9 · Motorola (1981), Appendix A, ADDD · Astle, LWTools 4.25, ch. 3.

ADDE

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 9B` · indexed `11 AB` · extended `11 BB` · immediate `11 8B`.
Add the operand to E. Keep the low 8 bits in E.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H records carry from bit 3.
Reading: Atkinson (2009), pp. 8 · Astle, LWTools 4.25, ch. 3.

ADDF

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 DB` · indexed `11 EB` · extended `11 FB` · immediate `11 CB`.
Add the operand to F. Keep the low 8 bits in F.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H records carry from bit 3.
Reading: Atkinson (2009), pp. 8 · Astle, LWTools 4.25, ch. 3.

ADDR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 30`.
Add source to destination.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 10 · Astle, LWTools 4.25, ch. 3.

ADDW

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 9B` · indexed `10 AB` · extended `10 BB` · immediate `10 8B`.
Add the operand to W. Keep the low 16 bits in W.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks unsigned carry. H is preserved.
Reading: Atkinson (2009), pp. 9 · Astle, LWTools 4.25, ch. 3.

DAA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `19`.
Adjust A after adding packed BCD bytes, using H and C from the addition.
Flags. N reflects the sign bit. Z marks a zero result. C reports decimal carry. V is undefined.
Reading: Atkinson (2009), pp. 56 · Motorola (1981), Appendix A, DAA · Astle, LWTools 4.25, ch. 3.

DEC

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `0A` · indexed `6A` · extended `7A`.
Operate on a memory byte. Subtract one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Motorola (1981), Appendix A, DEC · Astle, LWTools 4.25, ch. 3.

DECA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `4A`.
Operate on register A. Subtract one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Motorola (1981), Appendix A, DECA · Astle, LWTools 4.25, ch. 3.

DECB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `5A`.
Operate on register B. Subtract one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Motorola (1981), Appendix A, DECB · Astle, LWTools 4.25, ch. 3.

DECD

HD6309 only

```
No operand
```

Opcode bytes. `10 4A`.
Operate on register D. Subtract one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Astle, LWTools 4.25, ch. 3.

DECE

HD6309 only

```
No operand
```

Opcode bytes. `11 4A`.
Operate on register E. Subtract one.

Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Astle, LWTools 4.25, ch. 3.

DECF

HD6309 only

```
No operand
```

Opcode bytes. `11 5A`.
Operate on register F. Subtract one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Astle, LWTools 4.25, ch. 3.

DECW

HD6309 only

```
No operand
```

Opcode bytes. `10 5A`.
Operate on register W. Subtract one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from minimum negative to maximum positive. C is preserved.
Reading: Atkinson (2009), pp. 57–58 · Astle, LWTools 4.25, ch. 3.

DIVD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 9D` · indexed `11 AD` · extended `11 BD` · immediate `11 8D`.
Divide signed D by a signed byte. The quotient goes in B and the remainder in A. A zero divisor traps.
Flags. For an in-range quotient, N and Z describe it, V clears, and C receives its low bit. Signed overflow sets V. Range overflow preserves the dividend and clears N, Z and C.
Reading: Atkinson (2009), pp. 59 · Astle, LWTools 4.25, ch. 3.

DIVQ

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 9E` · indexed `11 AE` · extended `11 BE` · immediate `11 8E`.
Divide signed Q by a signed word. The quotient goes in W and the remainder in D. A zero divisor traps.
Flags. For an in-range quotient, N and Z describe it, V clears, and C receives its low bit. Signed overflow sets V. Range overflow preserves the dividend and clears N, Z and C.
VCC Plus's retained core has differences in DIVQ overflow and carry handling. Treat these flags as processor documentation.
Reading: Atkinson (2009), pp. 60 · Astle, LWTools 4.25, ch. 3.

INC

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `0C` · indexed `6C` · extended `7C`.
Operate on a memory byte. Add one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Motorola (1981), Appendix A, INC · Astle, LWTools 4.25, ch. 3.

INCA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `4C`.
Operate on register A. Add one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Motorola (1981), Appendix A, INCA · Astle, LWTools 4.25, ch. 3.

INCB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `5C`.
Operate on register B. Add one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Motorola (1981), Appendix A, INCB · Astle, LWTools 4.25, ch. 3.

INCD

HD6309 only

```
No operand
```

Opcode bytes. `10 4C`.
Operate on register D. Add one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Astle, LWTools 4.25, ch. 3.

INCE

HD6309 only

```
No operand
```

Opcode bytes. `11 4C`.
Operate on register E. Add one.

Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Astle, LWTools 4.25, ch. 3.

INCF

HD6309 only

```
No operand
```

Opcode bytes. `11 5C`.
Operate on register F. Add one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Astle, LWTools 4.25, ch. 3.

INCW

HD6309 only

```
No operand
```

Opcode bytes. `10 5C`.
Operate on register W. Add one.
Flags. N reflects the sign bit. Z marks a zero result. V sets on crossing from maximum positive to minimum negative. C is preserved.
Reading: Atkinson (2009), pp. 67–68 · Astle, LWTools 4.25, ch. 3.

MUL

MC6809 and HD6309

```
No operand
```

Opcode bytes. `3D`.
Multiply unsigned A by B and place the 16-bit product in D.
Flags. Z marks a zero product. C receives product bit 7.
Reading: Atkinson (2009), pp. 100 · Motorola (1981), Appendix A, MUL · Astle, LWTools 4.25, ch. 3.

MULD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 9F` · indexed `11 AF` · extended `11 BF` · immediate `11 8F`.
Multiply signed D by a signed 16-bit operand and store the 32-bit product in Q.
Flags. N reflects the sign bit. Z marks a zero result. V and C clear.
Reading: Atkinson (2009), pp. 101 · Astle, LWTools 4.25, ch. 3.

NEG

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `00` · indexed `60` · extended `70`.
Operate on a memory byte. Subtract the value from zero.
Flags. N reflects the sign bit. Z marks a zero result. C sets for a nonzero input. V marks negation of the most negative value.
Reading: Atkinson (2009), pp. 102–103 · Motorola (1981), Appendix A, NEG · Astle, LWTools 4.25, ch. 3.

NEGA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `40`.
Operate on register A. Subtract the value from zero.
Flags. N reflects the sign bit. Z marks a zero result. C sets for a nonzero input. V marks negation of the most negative value.
Reading: Atkinson (2009), pp. 102–103 · Motorola (1981), Appendix A, NEGA · Astle, LWTools 4.25, ch. 3.

NEGB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `50`.
Operate on register B. Subtract the value from zero.
Flags. N reflects the sign bit. Z marks a zero result. C sets for a nonzero input. V marks negation of the most negative value.
Reading: Atkinson (2009), pp. 102–103 · Motorola (1981), Appendix A, NEGB · Astle, LWTools 4.25, ch. 3.

NEGD

HD6309 only

```
No operand
```

Opcode bytes. `10 40`.
Operate on register D. Subtract the value from zero.
Flags. N reflects the sign bit. Z marks a zero result. C sets for a nonzero input. V marks negation of the most negative value.
Reading: Atkinson (2009), pp. 102–103 · Astle, LWTools 4.25, ch. 3.

SBCA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `92` · indexed `A2` · extended `B2` · immediate `82`.
Subtract the operand and incoming borrow from A. Write the difference to A.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 122 · Motorola (1981), Appendix A, SBCA · Astle, LWTools 4.25, ch. 3.

SBCB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D2` · indexed `E2` · extended `F2` · immediate `C2`.
Subtract the operand and incoming borrow from B. Write the difference to B.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 122 · Motorola (1981), Appendix A, SBCB · Astle, LWTools 4.25, ch. 3.

SBCD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 92` · indexed `10 A2` · extended `10 B2` · immediate `10 82`.
Subtract the operand and incoming borrow from D. Write the difference to D.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 123 · Astle, LWTools 4.25, ch. 3.

SBCR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 33`.
Subtract source and borrow from destination.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 124 · Astle, LWTools 4.25, ch. 3.

SEX

MC6809 and HD6309

```
No operand
```

Opcode bytes. `1D`.
Copy B's sign bit across A to extend B into D.
Flags. N reflects the sign bit. Z marks a zero result. V and C are preserved.
Reading: Atkinson (2009), pp. 125 · Motorola (1981), Appendix A, SEX · Astle, LWTools 4.25, ch. 3.

SEXW

HD6309 only

```
No operand
```

Opcode bytes. `14`.
Copy W's sign bit across D to extend W into Q.
Flags. N reflects the sign bit. Z marks a zero result. V and C are preserved.
Reading: Atkinson (2009), pp. 126 · Astle, LWTools 4.25, ch. 3.

SUBA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `90` · indexed `A0` · extended `B0` · immediate `80`.
Subtract the operand from A. Write the difference to A.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 131 · Motorola (1981), Appendix A, SUBA · Astle, LWTools 4.25, ch. 3.

SUBB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D0` · indexed `E0` · extended `F0` · immediate `C0`.
Subtract the operand from B. Write the difference to B.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 131 · Motorola (1981), Appendix A, SUBB · Astle, LWTools 4.25, ch. 3.

SUBD

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `93` · indexed `A3` · extended `B3` · immediate `83`.
Subtract the operand from D. Write the difference to D.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 132 · Motorola (1981), Appendix A, SUBD · Astle, LWTools 4.25, ch. 3.

SUBE

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 90` · indexed `11 A0` · extended `11 B0` · immediate `11 80`.
Subtract the operand from E. Write the difference to E.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 131 · Astle, LWTools 4.25, ch. 3.

SUBF

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 D0` · indexed `11 E0` · extended `11 F0` · immediate `11 C0`.
Subtract the operand from F. Write the difference to F.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 131 · Astle, LWTools 4.25, ch. 3.

SUBR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 32`.
Subtract source from destination.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 133 · Astle, LWTools 4.25, ch. 3.

SUBW

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 90` · indexed `10 A0` · extended `10 B0` · immediate `10 80`.
Subtract the operand from W. Write the difference to W.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 132 · Astle, LWTools 4.25, ch. 3.

Test and change bits

In the lab, see which value controls a pixel.
`AIM` · `ANDA` · `ANDB` · `ANDD` · `ANDR` · `BAND` · `BEOR` · `BIAND` · `BIEOR` · `BIOR` · `BITA` · `BITB` · `BITD` · `BOR` · `CLR` · `CLRA` · `CLRB` · `CLRD` · `CLRE` · `CLRF` · `CLRW` · `COM` · `COMA` · `COMB` · `COMD` · `COME` · `COMF` · `COMW` · `EIM` · `EORA` · `EORB` · `EORD` · `EORR` · `LDBT` · `OIM` · `ORA` · `ORB` · `ORD` · `ORR` · `STBT` · `TIM`

AIM

HD6309 only

```
#mask,address   ·   #mask,offset,X
```

Opcode bytes. direct `02` · indexed `62` · extended `72`.
Combine the literal mask and the addressed byte with AND. Write the result to that address.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 11 · Astle, LWTools 4.25, ch. 3.

ANDA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `94` · indexed `A4` · extended `B4` · immediate `84`.
Keep bits set in both inputs. Combine A with the operand and save the result in A.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 12 · Motorola (1981), Appendix A, ANDA · Astle, LWTools 4.25, ch. 3.

ANDB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D4` · indexed `E4` · extended `F4` · immediate `C4`.
Keep bits set in both inputs. Combine B with the operand and save the result in B.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 12 · Motorola (1981), Appendix A, ANDB · Astle, LWTools 4.25, ch. 3.

ANDD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 94` · indexed `10 A4` · extended `10 B4` · immediate `10 84`.
Keep bits set in both inputs. Combine D with the operand and save the result in D.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 14 · Astle, LWTools 4.25, ch. 3.

ANDR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 34`.
Apply bitwise AND to the destination using the source.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 15 · Astle, LWTools 4.25, ch. 3.

BAND

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 30`.
Combine the selected memory bit with the destination register bit using AND. Only the destination bit changes.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 20 · Astle, LWTools 4.25, ch. 3.

BEOR

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 34`.
Combine the selected memory bit with the destination register bit using exclusive OR. Only the destination bit changes.

Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 23 · Astle, LWTools 4.25, ch. 3.

BIAND

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 31`.
Combine the selected memory bit with the destination register bit using AND with the inverted source. Only the destination bit changes.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 29 · Astle, LWTools 4.25, ch. 3.

BIEOR

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 35`.
Combine the selected memory bit with the destination register bit using exclusive OR with the inverted source. Only the destination bit changes.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 30 · Astle, LWTools 4.25, ch. 3.

BIOR

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 33`.
Combine the selected memory bit with the destination register bit using OR with the inverted source. Only the destination bit changes.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 31 · Astle, LWTools 4.25, ch. 3.

BITA

MC6809 and HD6309

```
#literal  ·  <direct  ·  offset,X  ·  >address
```

Opcode bytes. direct `95` · indexed `A5` · extended `B5` · immediate `85`.
Test the bits of A selected by the operand. Preserve both inputs.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 32 · Motorola (1981), Appendix A, BITA · Astle, LWTools 4.25, ch. 3.

BITB

MC6809 and HD6309

```
#literal  ·  <direct  ·  offset,X  ·  >address
```

Opcode bytes. direct `D5` · indexed `E5` · extended `F5` · immediate `C5`.
Test the bits of B selected by the operand. Preserve both inputs.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 32 · Motorola (1981), Appendix A, BITB · Astle, LWTools 4.25, ch. 3.

BITD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 95` · indexed `10 A5` · extended `10 B5` · immediate `10 85`.
Test the bits of D selected by the operand. Preserve both inputs.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 33 · Astle, LWTools 4.25, ch. 3.

BOR

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 32`.
Combine the selected memory bit with the destination register bit using OR. Only the destination bit changes.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 41 · Astle, LWTools 4.25, ch. 3.

CLR

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `0F` · indexed `6F` · extended `7F`.
Operate on a memory byte. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Motorola (1981), Appendix A, CLR · Astle, LWTools 4.25, ch. 3.

CLRA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `4F`.
Operate on register A. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Motorola (1981), Appendix A, CLRA · Astle, LWTools 4.25, ch. 3.

CLRB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `5F`.

Operate on register B. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Motorola (1981), Appendix A, CLRB · Astle, LWTools 4.25, ch. 3.

CLRD

HD6309 only

```
No operand
```

Opcode bytes. `10 4F`.
Operate on register D. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Astle, LWTools 4.25, ch. 3.

CLRE

HD6309 only

```
No operand
```

Opcode bytes. `11 4F`.
Operate on register E. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Astle, LWTools 4.25, ch. 3.

CLRF

HD6309 only

```
No operand
```

Opcode bytes. `11 5F`.
Operate on register F. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Astle, LWTools 4.25, ch. 3.

CLRW

HD6309 only

```
No operand
```

Opcode bytes. `10 5F`.
Operate on register W. Write zero.
Flags. Z sets. N, V and C clear.
Reading: Atkinson (2009), pp. 48–49 · Astle, LWTools 4.25, ch. 3.

COM

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `03` · indexed `63` · extended `73`.
Operate on a memory byte. Invert every bit.

Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.
Reading: Atkinson (2009), pp. 53–54 · Motorola (1981), Appendix A, COM · Astle, LWTools 4.25, ch. 3.

COMA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `43`.
Operate on register A. Invert every bit.
Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.
Reading: Atkinson (2009), pp. 53–54 · Motorola (1981), Appendix A, COMA · Astle, LWTools 4.25, ch. 3.

COMB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `53`.
Operate on register B. Invert every bit.
Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.
Reading: Atkinson (2009), pp. 53–54 · Motorola (1981), Appendix A, COMB · Astle, LWTools 4.25, ch. 3.

COMD

HD6309 only

```
No operand
```

Opcode bytes. `10 43`.
Operate on register D. Invert every bit.
Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.
Reading: Atkinson (2009), pp. 53–54 · Astle, LWTools 4.25, ch. 3.

COME

HD6309 only

```
No operand
```

Opcode bytes. `11 43`.
Operate on register E. Invert every bit.
Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.
Reading: Atkinson (2009), pp. 53–54 · Astle, LWTools 4.25, ch. 3.

COMF

HD6309 only

```
No operand
```

Opcode bytes. `11 53`.
Operate on register F. Invert every bit.
Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.

Reading: Atkinson (2009), pp. 53–54 · Astle, LWTools 4.25, ch. 3.

COMW

HD6309 only

```
No operand
```

Opcode bytes. `10 53`.
Operate on register W. Invert every bit.
Flags. N reflects the sign bit. Z marks a zero result. C sets and V clears.
Reading: Atkinson (2009), pp. 53–54 · Astle, LWTools 4.25, ch. 3.

EIM

HD6309 only

```
#mask,address   ·   #mask,offset,X
```

Opcode bytes. direct `05` · indexed `65` · extended `75`.
Combine the literal mask and the addressed byte with exclusive OR. Write the result to that address.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 61 · Astle, LWTools 4.25, ch. 3.

EORA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `98` · indexed `A8` · extended `B8` · immediate `88`.
Set bits where the inputs differ. Combine A with the operand and save the result in A.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 62 · Motorola (1981), Appendix A, EORA · Astle, LWTools 4.25, ch. 3.

EORB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D8` · indexed `E8` · extended `F8` · immediate `C8`.
Set bits where the inputs differ. Combine B with the operand and save the result in B.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 62 · Motorola (1981), Appendix A, EORB · Astle, LWTools 4.25, ch. 3.

EORD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 98` · indexed `10 A8` · extended `10 B8` · immediate `10 88`.
Set bits where the inputs differ. Combine D with the operand and save the result in D.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 63 · Astle, LWTools 4.25, ch. 3.

EORR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 36`.
Toggle destination bits selected by the source.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 64 · Astle, LWTools 4.25, ch. 3.

LDBT

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 36`.
Combine the selected memory bit with the destination register bit using copy. Only the destination bit changes.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 92 · Astle, LWTools 4.25, ch. 3.

OIM

HD6309 only

```
#mask,address   ·   #mask,offset,X
```

Opcode bytes. direct `01` · indexed `61` · extended `71`.
Combine the literal mask and the addressed byte with OR. Write the result to that address.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 105 · Astle, LWTools 4.25, ch. 3.

ORA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `9A` · indexed `AA` · extended `BA` · immediate `8A`.
Set bits present in either input. Combine A with the operand and save the result in A.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 106 · Motorola (1981), Appendix A, ORA · Astle, LWTools 4.25, ch. 3.

ORB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `DA` · indexed `EA` · extended `FA` · immediate `CA`.
Set bits present in either input. Combine B with the operand and save the result in B.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.

Reading: Atkinson (2009), pp. 106 · Motorola (1981), Appendix A, ORB · Astle, LWTools 4.25, ch. 3.

ORD

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 9A` · indexed `10 AA` · extended `10 BA` · immediate `10 8A`.
Set bits present in either input. Combine D with the operand and save the result in D.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 108 · Astle, LWTools 4.25, ch. 3.

ORR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 35`.
Apply bitwise OR to the destination using the source.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 109 · Astle, LWTools 4.25, ch. 3.

STBT

HD6309 only

```
A|B|CC,source bit,destination bit,direct address
```

Opcode bytes. `11 37`.
Copy the selected register bit to the destination bit in direct-page memory.
Flags. CC can change when a destination bit in CC is selected.
Reading: Atkinson (2009), pp. 129 · Astle, LWTools 4.25, ch. 3.

TIM

HD6309 only

```
#mask,address   ·   #mask,offset,X
```

Opcode bytes. direct `0B` · indexed `6B` · extended `7B`.
Combine the literal mask and the addressed byte with AND. Use the result only to test flags.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 139 · Astle, LWTools 4.25, ch. 3.

Shift and rotate

In the lab, interpret the high bit and carry.

`ASL` · `ASLA` · `ASLB` · `ASLD` · `ASR` · `ASRA` · `ASRB` · `ASRD` · `LSL` · `LSLA` · `LSLB` · `LSLD` · `LSR` · `LSRA` · `LSRB` · `LSRD` · `LSRW` · `ROL` · `ROLA` · `ROLB` · `ROLD` · `ROLW` · `ROR` · `RORA` · `RORB` · `RORD` · `RORW`

ASL

Other spellings: `LSL`.

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `08` · indexed `68` · extended `78`.

Operate on a memory byte. Shift one place left and fill bit 0 with zero.

Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C. H is undefined.

Reading: Atkinson (2009), pp. 16–17 · Motorola (1981), Appendix A, ASL · Astle, LWTools 4.25, ch. 3.

ASLA

Other spellings: `LSLA`.

MC6809 and HD6309

```
No operand
```

Opcode bytes. `48`.

Operate on register A. Shift one place left and fill bit 0 with zero.

Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C. H is undefined.

Reading: Atkinson (2009), pp. 16–17 · Motorola (1981), Appendix A, ASLA · Astle, LWTools 4.25, ch. 3.

ASLB

Other spellings: `LSLB`.

MC6809 and HD6309

```
No operand
```

Opcode bytes. `58`.

Operate on register B. Shift one place left and fill bit 0 with zero.

Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C. H is undefined.

Reading: Atkinson (2009), pp. 16–17 · Motorola (1981), Appendix A, ASLB · Astle, LWTools 4.25, ch. 3.

ASLD

Other spellings: `LSLD`.

HD6309 only

```
No operand
```

Opcode bytes. `10 48`.

Operate on register D. Shift one place left and fill bit 0 with zero.

Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C.

Reading: Atkinson (2009), pp. 16–17 · Astle, LWTools 4.25, ch. 3.

ASR

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `07` · indexed `67` · extended `77`.

Operate on a memory byte. Shift right, preserving the sign bit.

Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved. H is undefined.
Reading: Atkinson (2009), pp. 18–19 · Motorola (1981), Appendix A, ASR · Astle, LWTools 4.25, ch. 3.

ASRA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `47`.
Operate on register A. Shift right, preserving the sign bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved. H is undefined.
Reading: Atkinson (2009), pp. 18–19 · Motorola (1981), Appendix A, ASRA · Astle, LWTools 4.25, ch. 3.

ASRB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `57`.
Operate on register B. Shift right, preserving the sign bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved. H is undefined.
Reading: Atkinson (2009), pp. 18–19 · Motorola (1981), Appendix A, ASRB · Astle, LWTools 4.25, ch. 3.

ASRD

HD6309 only

```
No operand
```

Opcode bytes. `10 47`.
Operate on register D. Shift right, preserving the sign bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 18–19 · Astle, LWTools 4.25, ch. 3.

LSR

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `04` · indexed `64` · extended `74`.
Operate on a memory byte. Shift right, bringing zero into the top bit.
Flags. N clears. Z marks zero. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 98–99 · Motorola (1981), Appendix A, LSR · Astle, LWTools 4.25, ch. 3.

LSRA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `44`.
Operate on register A. Shift right, bringing zero into the top bit.
Flags. N clears. Z marks zero. C receives the old bit 0. V is preserved.

Reading: Atkinson (2009), pp. 98–99 · Motorola (1981), Appendix A, LSRA · Astle, LWTools 4.25, ch. 3.

LSRB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `54`.
Operate on register B. Shift right, bringing zero into the top bit.
Flags. N clears. Z marks zero. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 98–99 · Motorola (1981), Appendix A, LSRB · Astle, LWTools 4.25, ch. 3.

LSRD

HD6309 only

```
No operand
```

Opcode bytes. `10 44`.
Operate on register D. Shift right, bringing zero into the top bit.
Flags. N clears. Z marks zero. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 98–99 · Astle, LWTools 4.25, ch. 3.

LSRW

HD6309 only

```
No operand
```

Opcode bytes. `10 54`.
Operate on register W. Shift right, bringing zero into the top bit.
Flags. N clears. Z marks zero. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 98–99 · Astle, LWTools 4.25, ch. 3.

ROL

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `09` · indexed `69` · extended `79`.
Operate on a memory byte. Shift left, moving the old carry into bit 0.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C.
Reading: Atkinson (2009), pp. 116–117 · Motorola (1981), Appendix A, ROL · Astle, LWTools 4.25, ch. 3.

ROLA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `49`.
Operate on register A. Shift left, moving the old carry into bit 0.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C.
Reading: Atkinson (2009), pp. 116–117 · Motorola (1981), Appendix A, ROLA · Astle, LWTools 4.25, ch. 3.

ROLB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `59`.
Operate on register B. Shift left, moving the old carry into bit 0.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C.
Reading: Atkinson (2009), pp. 116–117 · Motorola (1981), Appendix A, ROLB · Astle, LWTools 4.25, ch. 3.

ROLD

HD6309 only

```
No operand
```

Opcode bytes. `10 49`.
Operate on register D. Shift left, moving the old carry into bit 0.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C.
Reading: Atkinson (2009), pp. 116–117 · Astle, LWTools 4.25, ch. 3.

ROLW

HD6309 only

```
No operand
```

Opcode bytes. `10 59`.
Operate on register W. Shift left, moving the old carry into bit 0.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old top bit. V is N XOR C.
Reading: Atkinson (2009), pp. 116–117 · Astle, LWTools 4.25, ch. 3.

ROR

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `06` · indexed `66` · extended `76`.
Operate on a memory byte. Shift right, moving the old carry into the top bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 118–119 · Motorola (1981), Appendix A, ROR · Astle, LWTools 4.25, ch. 3.

RORA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `46`.
Operate on register A. Shift right, moving the old carry into the top bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 118–119 · Motorola (1981), Appendix A, RORA · Astle, LWTools 4.25, ch. 3.

RORB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `56`.
Operate on register B. Shift right, moving the old carry into the top bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 118–119 · Motorola (1981), Appendix A, RORB · Astle, LWTools 4.25, ch. 3.

RORD

HD6309 only

```
No operand
```

Opcode bytes. `10 46`.
Operate on register D. Shift right, moving the old carry into the top bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 118–119 · Astle, LWTools 4.25, ch. 3.

RORW

HD6309 only

```
No operand
```

Opcode bytes. `10 56`.
Operate on register W. Shift right, moving the old carry into the top bit.
Flags. N reflects the sign bit. Z marks a zero result. C receives the old bit 0. V is preserved.
Reading: Atkinson (2009), pp. 118–119 · Astle, LWTools 4.25, ch. 3.

Compare and choose a path

In the lab, test signed and unsigned decisions.
`BCC` · `BCS` · `BEQ` · `BGE` · `BGT` · `BHI` · `BHS` · `BLE` · `BLO` · `BLS` · `BLT` · `BMI` · `BNE` · `BPL` · `BRA` · `BRN` · `BVC` · `BVS` · `CMPA` · `CMPB` · `CMPD` · `CMPE` · `CMPF` · `CMPR` · `CMPS` · `CMPU` · `CMPW` · `CMPX` · `CMPY` · `JMP` · `LBCC` · `LBCS` · `LBEQ` · `LBGE` · `LBGT` · `LBHI` · `LBHS` · `LBLE` · `LBLO` · `LBLS` · `LBLT` · `LBMI` · `LBNE` · `LBPL` · `LBRA` · `LBRN` · `LBVC` · `LBVS` · `TST` · `TSTA` · `TSTB` · `TSTD` · `TSTE` · `TSTF` · `TSTW`

BCC

Other spellings: `BHS`.
MC6809 and HD6309

```
target label
```

Opcode bytes. `24`.
Continue at the target when C is clear (unsigned higher or equal after a compare). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 21 · Motorola (1981), Appendix A, BCC · Astle, LWTools 4.25, ch. 3.

BCS

Other spellings: `BLO`.

MC6809 and HD6309

```
target label
```

Opcode bytes. 25.
Continue at the target when C is set (unsigned lower). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 22 · Motorola (1981), Appendix A, BCS · Astle, LWTools 4.25, ch. 3.

BEQ

MC6809 and HD6309

```
target label
```

Opcode bytes. 27.
Continue at the target when Z is set (equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 24 · Motorola (1981), Appendix A, BEQ · Astle, LWTools 4.25, ch. 3.

BGE

MC6809 and HD6309

```
target label
```

Opcode bytes. 2C.
Continue at the target when N equals V (signed greater or equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 25 · Motorola (1981), Appendix A, BGE · Astle, LWTools 4.25, ch. 3.

BGT

MC6809 and HD6309

```
target label
```

Opcode bytes. 2E.
Continue at the target when Z is clear and N equals V (signed greater). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 26 · Motorola (1981), Appendix A, BGT · Astle, LWTools 4.25, ch. 3.

BHI

MC6809 and HD6309

```
target label
```

Opcode bytes. 22.

Continue at the target when C and Z are both clear (unsigned higher). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 27 · Motorola (1981), Appendix A, BHI · Astle, LWTools 4.25, ch. 3.

BLE

MC6809 and HD6309

```
target label
```

Opcode bytes. 2F.
Continue at the target when Z is set or N differs from V (signed less or equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 35 · Motorola (1981), Appendix A, BLE · Astle, LWTools 4.25, ch. 3.

BLS

MC6809 and HD6309

```
target label
```

Opcode bytes. 23.
Continue at the target when C or Z is set (unsigned lower or equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 37 · Motorola (1981), Appendix A, BLS · Astle, LWTools 4.25, ch. 3.

BLT

MC6809 and HD6309

```
target label
```

Opcode bytes. 2D.
Continue at the target when N differs from V (signed less). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 38 · Motorola (1981), Appendix A, BLT · Astle, LWTools 4.25, ch. 3.

BMI

MC6809 and HD6309

```
target label
```

Opcode bytes. 2B.
Continue at the target when N is set (negative). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 39 · Motorola (1981), Appendix A, BMI · Astle, LWTools 4.25, ch. 3.

BNE

MC6809 and HD6309

```
target label
```

Opcode bytes. 26.
Continue at the target when Z is clear (unequal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 40 · Motorola (1981), Appendix A, BNE · Astle, LWTools 4.25, ch. 3.

BPL

MC6809 and HD6309

```
target label
```

Opcode bytes. 2A.
Continue at the target when N is clear (nonnegative). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 42 · Motorola (1981), Appendix A, BPL · Astle, LWTools 4.25, ch. 3.

BRA

MC6809 and HD6309

```
target label
```

Opcode bytes. 20.
Continue at the target. A branch to its own label forms a hold loop.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 43 · Motorola (1981), Appendix A, BRA · Astle, LWTools 4.25, ch. 3.

BRN

MC6809 and HD6309

```
target label
```

Opcode bytes. 21.
Read the displacement and continue at the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 44 · Motorola (1981), Appendix A, BRN · Astle, LWTools 4.25, ch. 3.

BVC

MC6809 and HD6309

```
target label
```

Opcode bytes. 28.
Continue at the target when V is clear (no signed overflow). Otherwise execute the next instruction.

Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 46 · Motorola (1981), Appendix A, BVC · Astle, LWTools 4.25, ch. 3.

BVS

MC6809 and HD6309

```
target label
```

Opcode bytes. 29.
Continue at the target when V is set (signed overflow). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 47 · Motorola (1981), Appendix A, BVS · Astle, LWTools 4.25, ch. 3.

CMPA

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `91` · indexed `A1` · extended `B1` · immediate `81`.
Subtract the operand from A. Keep A intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 50 · Motorola (1981), Appendix A, CMPA · Astle, LWTools 4.25, ch. 3.

CMPB

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `D1` · indexed `E1` · extended `F1` · immediate `C1`.
Subtract the operand from B. Keep B intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 50 · Motorola (1981), Appendix A, CMPB · Astle, LWTools 4.25, ch. 3.

CMPD

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 93` · indexed `10 A3` · extended `10 B3` · immediate `10 83`.
Subtract the operand from D. Keep D intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 51 · Motorola (1981), Appendix A, CMPD · Astle, LWTools 4.25, ch. 3.

CMPE

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 91` · indexed `11 A1` · extended `11 B1` · immediate `11 81`.

Subtract the operand from E. Keep E intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 50 · Astle, LWTools 4.25, ch. 3.

CMPF

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 D1` · indexed `11 E1` · extended `11 F1` · immediate `11 C1`.
Subtract the operand from F. Keep F intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow. H is undefined.
Reading: Atkinson (2009), pp. 50 · Astle, LWTools 4.25, ch. 3.

CMPR

HD6309 only

```
source register,destination register
```

Opcode bytes. `10 37`.
Compare destination minus source, preserving both.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
The destination selects the width. A word source supplies its low byte to a byte destination. For a word destination A/B supply D, E/F supply W, CC is zero-extended and DP supplies the high byte. PC is undefined in this family. See Atkinson, p. 143.
Reading: Atkinson (2009), pp. 52 · Astle, LWTools 4.25, ch. 3.

CMPS

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 9C` · indexed `11 AC` · extended `11 BC` · immediate `11 8C`.
Subtract the operand from S. Keep S intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 51 · Motorola (1981), Appendix A, CMPS · Astle, LWTools 4.25, ch. 3.

CMPU

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `11 93` · indexed `11 A3` · extended `11 B3` · immediate `11 83`.
Subtract the operand from U. Keep U intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 51 · Motorola (1981), Appendix A, CMPU · Astle, LWTools 4.25, ch. 3.

CMPW

HD6309 only

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 91` · indexed `10 A1` · extended `10 B1` · immediate `10 81`.
Subtract the operand from W. Keep W intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 51 · Astle, LWTools 4.25, ch. 3.

CMPX

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `9C` · indexed `AC` · extended `BC` · immediate `8C`.
Subtract the operand from X. Keep X intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 51 · Motorola (1981), Appendix A, CMPX · Astle, LWTools 4.25, ch. 3.

CMPY

MC6809 and HD6309

```
#literal   ·   <direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `10 9C` · indexed `10 AC` · extended `10 BC` · immediate `10 8C`.
Subtract the operand from Y. Keep Y intact for a later branch.
Flags. N reflects the sign bit. Z marks a zero result. V marks signed overflow. C marks a borrow.
Reading: Atkinson (2009), pp. 51 · Motorola (1981), Appendix A, CMPY · Astle, LWTools 4.25, ch. 3.

JMP

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `0E` · indexed `6E` · extended `7E`.
Set PC to the effective address.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 69 · Motorola (1981), Appendix A, JMP · Astle, LWTools 4.25, ch. 3.

LBCC

Other spellings: `LBHS`.

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 24`.
Continue at the target when C is clear (unsigned higher or equal after a compare). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 71 · Motorola (1981), Appendix A, LBCC · Astle, LWTools 4.25, ch. 3.

LBCS

Other spellings: `LBL0`.

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 25`.
Continue at the target when C is set (unsigned lower). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 72 · Motorola (1981), Appendix A, LBCS · Astle, LWTools 4.25, ch. 3.

LBEQ

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 27`.
Continue at the target when Z is set (equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 73 · Motorola (1981), Appendix A, LBEQ · Astle, LWTools 4.25, ch. 3.

LBGE

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 2C`.
Continue at the target when N equals V (signed greater or equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 74 · Motorola (1981), Appendix A, LBGE · Astle, LWTools 4.25, ch. 3.

LBGT

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 2E`.
Continue at the target when Z is clear and N equals V (signed greater). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 75 · Motorola (1981), Appendix A, LBGT · Astle, LWTools 4.25, ch. 3.

LBHI

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 22`.

Continue at the target when C and Z are both clear (unsigned higher). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 76 · Motorola (1981), Appendix A, LBHI · Astle, LWTools 4.25, ch. 3.

LBLE

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 2F`.
Continue at the target when Z is set or N differs from V (signed less or equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 78 · Motorola (1981), Appendix A, LBLE · Astle, LWTools 4.25, ch. 3.

LBLS

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 23`.
Continue at the target when C or Z is set (unsigned lower or equal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 80 · Motorola (1981), Appendix A, LBLS · Astle, LWTools 4.25, ch. 3.

LBLT

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 2D`.
Continue at the target when N differs from V (signed less). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 81 · Motorola (1981), Appendix A, LBLT · Astle, LWTools 4.25, ch. 3.

LBMI

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 2B`.
Continue at the target when N is set (negative). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 82 · Motorola (1981), Appendix A, LBMI · Astle, LWTools 4.25, ch. 3.

LBNE

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 26`.
Continue at the target when Z is clear (unequal). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 83 · Motorola (1981), Appendix A, LBNE · Astle, LWTools 4.25, ch. 3.

LBPL

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 2A`.
Continue at the target when N is clear (nonnegative). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 84 · Motorola (1981), Appendix A, LBPL · Astle, LWTools 4.25, ch. 3.

LBRA

MC6809 and HD6309

```
target label
```

Opcode bytes. `16`.
Continue at the target. A branch to its own label forms a hold loop.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 85 · Motorola (1981), Appendix A, LBRA · Astle, LWTools 4.25, ch. 3.

LBRN

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 21`.
Read the displacement and continue at the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 86 · Motorola (1981), Appendix A, LBRN · Astle, LWTools 4.25, ch. 3.

LBVC

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 28`.
Continue at the target when V is clear (no signed overflow). Otherwise execute the next instruction.

Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 88 · Motorola (1981), Appendix A, LBVC · Astle, LWTools 4.25, ch. 3.

LBVS

MC6809 and HD6309

```
target label
```

Opcode bytes. `10 29`.
Continue at the target when V is set (signed overflow). Otherwise execute the next instruction.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 89 · Motorola (1981), Appendix A, LBVS · Astle, LWTools 4.25, ch. 3.

TST

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `0D` · indexed `6D` · extended `7D`.
Operate on a memory byte. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Motorola (1981), Appendix A, TST · Astle, LWTools 4.25, ch. 3.

TSTA

MC6809 and HD6309

```
No operand
```

Opcode bytes. `4D`.
Operate on register A. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Motorola (1981), Appendix A, TSTA · Astle, LWTools 4.25, ch. 3.

TSTB

MC6809 and HD6309

```
No operand
```

Opcode bytes. `5D`.
Operate on register B. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Motorola (1981), Appendix A, TSTB · Astle, LWTools 4.25, ch. 3.

TSTD

HD6309 only

```
No operand
```

Opcode bytes. `10 4D`.

Operate on register D. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Astle, LWTools 4.25, ch. 3.

TSTE

HD6309 only

```
No operand
```

Opcode bytes. `11 4D`.
Operate on register E. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Astle, LWTools 4.25, ch. 3.

TSTF

HD6309 only

```
No operand
```

Opcode bytes. `11 5D`.
Operate on register F. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Astle, LWTools 4.25, ch. 3.

TSTW

HD6309 only

```
No operand
```

Opcode bytes. `10 5D`.
Operate on register W. Test the value while preserving it.
Flags. N reflects the sign bit. Z marks a zero result. V clears. C is preserved.
Reading: Atkinson (2009), pp. 140–141 · Astle, LWTools 4.25, ch. 3.

Call, save and return

In the lab, follow a return address on S.
`BSR` · `JSR` · `LBSR` · `PSHS` · `PSHSW` · `PSHU` · `PSHUW` · `PULS` · `PULSW` · `PULU` · `PULUW` · `RTS`

BSR

MC6809 and HD6309

```
target label
```

Opcode bytes. `8D`.
Push the following instruction's address on S, then continue at the target. RTS returns to the caller.
Flags. CC is preserved.
The signed 8-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 45 · Motorola (1981), Appendix A, BSR · Astle, LWTools 4.25, ch. 3.

JSR

MC6809 and HD6309

```
<direct   ·   offset,X   ·   >address
```

Opcode bytes. direct `9D` · indexed `AD` · extended `BD`.
Push the return address on S, then set PC to the effective address.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 70 · Motorola (1981), Appendix A, JSR · Astle, LWTools 4.25, ch. 3.

LBSR

MC6809 and HD6309

```
target label
```

Opcode bytes. `17`.
Push the following instruction's address on S, then continue at the target. RTS returns to the caller.
Flags. CC is preserved.
The signed 16-bit displacement is relative to the following instruction.
Reading: Atkinson (2009), pp. 87 · Motorola (1981), Appendix A, LBSR · Astle, LWTools 4.25, ch. 3.

PSHS

MC6809 and HD6309

```
register[,register…]   ·   #mask
```

Opcode bytes. `34`.
Save selected registers on S. The stack grows toward lower addresses.
Flags. CC is preserved.
Select CC, A, B, DP, X, Y, U or PC. LWASM also accepts D for A and B. Hardware determines the register order.
Reading: Atkinson (2009), pp. 110 · Motorola (1981), Appendix A, PSHS · Astle, LWTools 4.25, ch. 3.

PSHSW

HD6309 only

```
No operand
```

Opcode bytes. `10 38`.
Push both bytes of W onto S, subtracting two from the pointer.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 111 · Astle, LWTools 4.25, ch. 3.

PSHU

MC6809 and HD6309

```
register[,register…]   ·   #mask
```

Opcode bytes. `36`.
Save selected registers on U. The stack grows toward lower addresses.
Flags. CC is preserved.
Select CC, A, B, DP, X, Y, S or PC. LWASM also accepts D for A and B. Hardware determines the register order.

Reading: Atkinson (2009), pp. 110 · Motorola (1981), Appendix A, PSHU · Astle, LWTools 4.25, ch. 3.

PSHUW

HD6309 only

```
No operand
```

Opcode bytes. `10 3A`.
Push both bytes of W onto U, subtracting two from the pointer.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 112 · Astle, LWTools 4.25, ch. 3.

PULS

MC6809 and HD6309

```
register[,register…]   ·   #mask
```

Opcode bytes. `35`.
Restore selected registers from S, advancing the stack pointer. Pulling PC transfers control.
Flags. Pulling CC replaces all its flags.
Select CC, A, B, DP, X, Y, U or PC. LWASM also accepts D for A and B. Hardware determines the register order.
Reading: Atkinson (2009), pp. 113 · Motorola (1981), Appendix A, PULS · Astle, LWTools 4.25, ch. 3.

PULSW

HD6309 only

```
No operand
```

Opcode bytes. `10 39`.
Pull two bytes into W from S, advancing the pointer by two.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 114 · Astle, LWTools 4.25, ch. 3.

PULU

MC6809 and HD6309

```
register[,register…]   ·   #mask
```

Opcode bytes. `37`.
Restore selected registers from U, advancing the stack pointer. Pulling PC transfers control.
Flags. Pulling CC replaces all its flags.
Select CC, A, B, DP, X, Y, S or PC. LWASM also accepts D for A and B. Hardware determines the register order.
Reading: Atkinson (2009), pp. 113 · Motorola (1981), Appendix A, PULU · Astle, LWTools 4.25, ch. 3.

PULUW

HD6309 only

```
No operand
```

Opcode bytes. `10 3B`.
Pull two bytes into W from U, advancing the pointer by two.

Flags. CC is preserved.
Reading: Atkinson (2009), pp. 115 · Astle, LWTools 4.25, ch. 3.

RTS

MC6809 and HD6309

```
No operand
```

Opcode bytes. `39`.
Pull the return address from S into PC and advance S by two.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 121 · Motorola (1981), Appendix A, RTS · Astle, LWTools 4.25, ch. 3.

Interrupts and processor control

In the lab, examine the saved interrupt frame.
`ANDCC` · `BITMD` · `CWAI` · `LDMD` · `NOP` · `ORCC` · `RTI` · `SWI` · `SWI2` · `SWI3` · `SYNC`

ANDCC

MC6809 and HD6309

```
#mask
```

Opcode bytes. `1C`.
Keep only CC bits selected by the mask.
Flags. Any CC bit can clear.
Reading: Atkinson (2009), pp. 13 · Motorola (1981), Appendix A, ANDCC · Astle, LWTools 4.25, ch. 3.

BITMD

HD6309 only

```
#mask
```

Opcode bytes. `11 3C`.
Test the masked trap status bits of MD. Reading a selected sticky error clears it.
Flags. Z sets if no selected error was pending.
Reading: Atkinson (2009), pp. 34 · Astle, LWTools 4.25, ch. 3.

CWAI

MC6809 and HD6309

```
#mask
```

Opcode bytes. `3C`.
AND CC with the mask, stack the full interrupt context on S, then wait for an eligible interrupt.
Flags. The mask updates CC. E sets for the stacked full context.
Reading: Atkinson (2009), pp. 55 · Motorola (1981), Appendix A, CWAI · Astle, LWTools 4.25, ch. 3.

LDMD

HD6309 only

```
#mask
```

Opcode bytes. `11 3D`.
Write the low two mode-control bits of MD.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 93 · Astle, LWTools 4.25, ch. 3.

NOP

MC6809 and HD6309

```
No operand
```

Opcode bytes. `12`.
Advance to the following instruction.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 104 · Motorola (1981), Appendix A, NOP · Astle, LWTools 4.25, ch. 3.

ORCC

MC6809 and HD6309

```
#mask
```

Opcode bytes. `1A`.
Set each CC bit selected by the mask.
Flags. Any CC bit can set.
Reading: Atkinson (2009), pp. 107 · Motorola (1981), Appendix A, ORCC · Astle, LWTools 4.25, ch. 3.

RTI

MC6809 and HD6309

```
No operand
```

Opcode bytes. `3B`.
Pull CC and the return state from S. Stacked E determines whether to restore a full frame.
Flags. Restores saved CC.
Reading: Atkinson (2009), pp. 120 · Motorola (1981), Appendix A, RTI · Astle, LWTools 4.25, ch. 3.

SWI

MC6809 and HD6309

```
No operand
```

Opcode bytes. `3F`.
Save a full interrupt frame on S and fetch the new PC from $FFFA.
Flags. E sets. I and F set after stacking.
Reading: Atkinson (2009), pp. 134 · Motorola (1981), Appendix A, SWI · Astle, LWTools 4.25, ch. 3.

SWI2

MC6809 and HD6309

```
No operand
```

Opcode bytes. `10 3F`.
Save a full interrupt frame on S and fetch the new PC from $FFF4.
Flags. E sets. I and F keep their values.
Reading: Atkinson (2009), pp. 134 · Motorola (1981), Appendix A, SWI · Astle, LWTools 4.25, ch. 3.

SWI3

MC6809 and HD6309

```
No operand
```

Opcode bytes. `11 3F`.
Save a full interrupt frame on S and fetch the new PC from $FFF2.
Flags. E sets. I and F keep their values.
Reading: Atkinson (2009), pp. 134 · Motorola (1981), Appendix A, SWI · Astle, LWTools 4.25, ch. 3.

SYNC

MC6809 and HD6309

```
No operand
```

Opcode bytes. `13`.
Wait for an interrupt request. Even a masked request can release the wait.
Flags. CC is preserved.
Reading: Atkinson (2009), pp. 135 · Motorola (1981), Appendix A, SYNC · Astle, LWTools 4.25, ch. 3.

Transfer a block

In the lab, inspect the destination and W.
`COPY` · `COPY+` · `COPY–` · `EXP` · `EXPAND` · `IMP` · `IMPLODE` · `TFM` · `TFRM` · `TFRP` · `TFRR` · `TFRS`

TFM

Other spellings: `COPY`, `COPY+`, `COPY–`, `TFRP`, `TFRM`, `IMP`, `IMPLODE`, `TFRS`, `EXP`, `EXPAND`, `TFRR`.
HD6309 only

```
r+,r+   ·   r–,r–   ·   r+,r   ·   r,r+
```

Opcode bytes. r+,r+ `11 38` · r-,r- `11 39` · r+,r `11 3A` · r,r+ `11 3B`.
Copy W bytes between addresses held in D, X, Y, U or S. Each suffix advances, retreats or holds its pointer after a byte. W counts down to zero. Without interruption the transfer costs 6 cycles plus 3 per byte.
Flags. CC is preserved.
LWASM's alternative spellings take two bare registers. COPY/COPY+/TFRP advance both, COPY-/TFRM retreat both, IMP/IMPLODE/TFRS advance the source and EXP/EXPAND/TFRR advance the destination. In VCC Plus, debugger Step finishes the block. Continue can hit a breakpoint again between bytes.
Reading: Atkinson (2009), pp. 136 · VCC Plus, Native Debugger, HD6309 Step · Astle, LWTools 4.25, ch. 3.

Registers, addressing and LWASM

Appendix B

Suppose the debugger says X points at the letter O. `LDA ,X+` reads O and advances X. `LEAX packet,PCR` finds the address of the letter and puts that address in X. The result in the register tells you which question you asked. Keep this page beside the receiver when one printed symbol changes the meaning of a line.

Read the register display

Look here	Use it for
A, B, D	Two 8-bit accumulators or one 16-bit value. A is the high byte of D. A change to A or B changes D immediately.
X, Y, U, S	Four 16-bit registers for indexed addressing. U and S can also address stacks. Hardware uses S on a subroutine call or interrupt.
PC, DP	Find the next instruction. Supply the upper address byte for a direct-page operand.
CC	Read the condition bits from high to low as E, F, H, I, N, Z, V, C. On a compare, read C for unsigned borrow and N together with V for signed order.
HD6309 E, F, W, Q	E and F form W, just as A and B form D. Q combines D above W.
HD6309 V, MD	V is another 16-bit register. MD controls the processor's execution mode and FIRQ frame choice and reports trap status.

The names E, F and V each appear twice in the Hitachi debugger. The accumulators E and F hold eight bits each, and the register V holds sixteen. The flags CC.E, CC.F and CC.V each hold one bit. For an interrupt return the saved **CC.E** answers whether an entire state frame was stacked. For a 6309 transfer the **E register** is the high half of the count in W. The interrupt exercise and the block-copy exercise let you see the distinction.

A

MC6809 and HD6309

Eight-bit accumulator. A forms the high byte of D.

Reading: Motorola (1981), ch. 1.

B

MC6809 and HD6309

Eight-bit accumulator. B forms the low byte of D.

Reading: Motorola (1981), ch. 1.

CC.C

MC6809 and HD6309

Carry or borrow. An unsigned addition sets C when the sum exceeds the register. A comparison sets C when the unsigned left operand is lower.

Reading: Motorola (1981), ch. 1.

CC.E

MC6809 and HD6309

Entire state stacked. RTI checks E in the saved condition byte to choose between the short and full interrupt frames.

Reading: Motorola (1981), ch. 1.

CC.F

MC6809 and HD6309

Fast interrupt request mask. Set F to mask FIRQ. A saved CC value can restore the previous mask.

Reading: Motorola (1981), ch. 1.

CC.H

MC6809 and HD6309

Half carry. An eight-bit addition places the carry from bit 3 here. DAA uses H to adjust a packed decimal result.

Reading: Motorola (1981), ch. 1.

CC.I

MC6809 and HD6309

Interrupt request mask. Set I to mask IRQ. Clear I to permit IRQ when the external request is active.

Reading: Motorola (1981), ch. 1.

CC.N

MC6809 and HD6309

Negative. For operations that set N from the result, it copies the most significant bit. A signed comparison also needs V.

Reading: Motorola (1981), ch. 1.

CC.V

MC6809 and HD6309

Signed overflow. For example, adding one to $7F produces $80 in a byte and sets V.

Reading: Motorola (1981), ch. 1.

CC.Z

MC6809 and HD6309

Zero. Many arithmetic and load instructions set Z when the result is zero. After a compare, BEQ uses Z to test equality.

Reading: Motorola (1981), ch. 1.

Condition codes

Other spellings: `CC`.

MC6809 and HD6309

CC holds eight one-bit values. E marks an entire interrupt frame. F masks FIRQ, H is half carry, I masks IRQ. N is negative, Z zero, V signed overflow and C carry or borrow.

Branch choices depend on how the previous instruction defined these bits. For example, Z=1 after CMP means equal; C=1 after CMP means unsigned lower. An undefined bit cannot support a portable decision.

Reading: Motorola (1981), ch. 1.

D

MC6809 and HD6309

Sixteen-bit accumulator built from A (high) and B (low). Writing D immediately changes both halves.

Reading: Motorola (1981), ch. 1.

DP

MC6809 and HD6309

Direct-page register. Its byte supplies the upper address bits of a direct operand. LWASM SETDP describes your assumption during assembly.

Reading: Motorola (1981), ch. 1.

E

HD6309 only

Eight-bit accumulator and high byte of the HD6309 W register. CC.E is a separate flag.

Reading: Burke (1993), register file.

F

HD6309 only

Eight-bit accumulator and low byte of W. CC.F is the fast-interrupt mask.

Reading: Burke (1993), register file.

HD6309 modes

HD6309 only

The HD6309 starts in emulation mode. Many instructions take fewer cycles in native mode. LDMD #1 selects native mode with the ordinary short FIRQ frame. Mode selection is independent of the LWASM target.

MD bit 0 selects native mode. Bit 1 selects full FIRQ stacking. A native full-state interrupt frame includes E and F; the mode on return must agree with the frame. Cycle counts also depend on indexed postbytes, branch outcomes and transfer lengths. In VCC Plus, Step completes a TFM instruction across its byte transfers.

Reading: Atkinson (2009), pp. 2, 120, 136, 144 · Burke (1993), mode register · VCC Plus, Native Debugger, HD6309 Step.

MD

HD6309 only

Mode register. LDMD controls execution mode and FIRQ stacking. BITMD tests the latched illegal-instruction and division-by-zero status.

Reading: Burke (1993), register file.

PC

MC6809 and HD6309

Program counter. Branches, jumps, calls and returns change the address of the next instruction.

Reading: Motorola (1981), ch. 1.

Q

HD6309 only

Thirty-two-bit accumulator. From most to least significant its bytes are A, B, E, F. D is the high word and W the low word.

Reading: Burke (1993), register file.

S

MC6809 and HD6309

Sixteen-bit hardware stack pointer. JSR, interrupts and PSHS use memory below S. Initialize a writable stack before making a standalone call.

Reading: Motorola (1981), ch. 1.

U

MC6809 and HD6309

Sixteen-bit user stack pointer, also available as an index register. PSHU and PULU use this stack.

Reading: Motorola (1981), ch. 1.

V

HD6309 only

Additional 16-bit HD6309 register used in register operations. CC.V is a separate overflow flag.
Reading: Burke (1993), register file.

W

HD6309 only

Sixteen-bit accumulator combining E (high) and F (low). TFM uses W as a byte count.
Reading: Burke (1993), register file.

X

MC6809 and HD6309

Sixteen-bit index register. Use X to address a table or traverse bytes.
Reading: Motorola (1981), ch. 1.

Y

MC6809 and HD6309

Sixteen-bit index register. X and Y can walk independent buffers.
Reading: Motorola (1981), ch. 1.

Decode an operand

Read the punctuation first. In these LWASM patterns, # selects the value itself; < or > requests a shorter or longer address encoding. An indexed operand describes how to form an address. Brackets add a pointer fetch. The instruction determines what happens to the resulting value or address.

Look at the receiver's source	What to check in the debugger
`LEAX packet,PCR`	X receives the assembled address of the packet. PCR lets the assembler encode its distance from the following instruction.
`ADDA ,X` followed by `LDA ,X+`	Both instructions read the same byte. Only the second advances X. Follow the saving and restoring of A around the copy.
`JSR [CHROUT]`	The two bytes at the CHROUT vector supply the subroutine address. At entry the saved return address is on S.
`BNE next`	The displacement is signed, measured from PC after the branch operand. This branch goes backward while the count is nonzero.

A CPU address selects memory *through* the current map. It identifies a location within a physical RAM block only when that block supplies the access. The MMU lesson shows how to check the active task, block and offset before drawing conclusions from a memory dump.

Direct

MC6809 and HD6309

```
LDA <$34
```

Join DP with the 8-bit operand to make the memory address. With DP=$12, LDA <$34 reads $1234.
Reading: Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

Extended

MC6809 and HD6309

```
LDA >$0400
```

Use the full 16-bit address in the instruction. LDA >$0400 reads a byte there.
On the CoCo 3, the address in STA >$1A00 selects a byte through the CPU memory map. The MMU chooses the physical 8 KiB block and preserves the low 13 address bits. Use Mapped CPU in the debugger to inspect that store.
Reading: Tepolt (1987), p. 20, fig. 3-4 · Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

HD6309 indexed

HD6309 only

```
LDA E,X   ·   LDA ,W++
```

E, F and W can supply offsets for X, Y, U or S. W also supports ,W, offset,W, ,W++ and ,--W, including their indirect forms.
Reading: Atkinson (2009), p. 148 · Astle, LWTools 4.25, ch. 3.

Immediate

MC6809 and HD6309

```
LDA #$20
```

Use the following operand bytes as the value.
Reading: Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

Indexed

MC6809 and HD6309

```
LDA ,X+   ·   LDD 2,S   ·   LDA B,Y
```

Calculate an address from a register and an offset. LDA ,X+ reads at the original X then increments X by one. Use ,X++ for a two-byte advance. Indexed operands can also use A, B or D as signed offsets.
Reading: Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

Indirect

MC6809 and HD6309

```
LDA [,X++]   ·   JMP [$4000]
```

Use square brackets to fetch a two-byte pointer at the intermediate address, then access the final address. Use a two-byte increment when walking a pointer table.
Reading: Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

Inherent

MC6809 and HD6309

```
INCA
```

The opcode specifies the operation and its operands. INCA, for example, acts on A.
Reading: Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

Relative

MC6809 and HD6309

```
BNE again   ·   LDA table,PCR
```

Add a signed displacement to PC after the operand. Branch targets are labels. Indexed PC-relative data references use PCR in LWASM.
Reading: Motorola (1981), ch. 2 · Astle, LWTools 4.25, ch. 3.

Assembler index

When you single-step, the processor fetches instruction bytes. `FCC "ORB"` produces three data bytes; `EQU` records a value the assembler can substitute later; `END start` supplies the execution address for this DECB build. None is a step the processor takes to construct the packet. The linked build shows how a symbol that names a location in one module receives its final CPU address.
Find the directive by its name or its alternative spelling in this book's search. Each entry gives the canonical LWASM syntax. Sections, imports, exports and relocatable object output belong together. Origin and entry directives for an absolute binary belong to a different build. A directive that reserves storage can advance the location counter without depositing a known byte; use an explicit initializer when the exercise depends on that byte's first value.

*PRAGMAPOP

```
*PRAGMAPOP
```

Restore saved pragma settings.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

*PRAGMAPUSH

```
*PRAGMAPUSH
```

Save pragma settings on the assembler's pragma stack.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ALIGN

```
ALIGN boundary
```

Advance to a boundary divisible by the supplied value.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

DEPHASE

```
DEPHASE
```

Resume the original address calculation after PHASE.
LWASM directive. Its effects take place during assembly.

Reading: Astle, LWTools 4.25, ch. 3.

DTB

```
DTB
```

Emit six host-local date and time bytes.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

DTS

```
DTS
```

Emit a textual build timestamp. Its value changes with the host clock.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ELSE

```
ELSE
```

Select the alternate branch of a conditional assembly block.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

END

Other spellings: `.END`.

```
END entry
```

Finish the assembly and specify the DECB execution address. END itself emits no machine instruction.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ENDC

Other spellings: `ENDIF`.

```
ENDC
```

Close the current conditional assembly block.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ENDM

```
ENDM
```

Finish a macro definition.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ENDS

Other spellings: `ENDSTRUCT`.

```
ENDS
```

Close a structure definition.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ENDSECT

Other spellings: `ENDSECTION`.

```
ENDSECT
```

Finish the current object section.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

EQU

Other spellings: =.

```
name EQU expression
```

Assign a fixed value to a symbol.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ERROR

```
ERROR message
```

Report an assembly error with the supplied text.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

EXPORT

Other spellings: `.GLOBL`.

```
EXPORT name
```

Expose a symbol to other object modules.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

EXTDEP

```
name EXTDEP
```

Mark an external symbol as a link dependency without emitting storage for it.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

EXTERN

Other spellings: `EXTERNAL`, `IMPORT`.

```
name EXTERN
```

Declare a name defined in another object module.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FCB

Other spellings: `.BYTE`, `.DB`.

```
FCB byte[,byte…]
```

Emit each expression as one byte. The listing shows data at the current location.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FCC

Other spellings: `.ASCII`, `.STR`.

```
FCC /text/
```

Emit character bytes between matching delimiters.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FCN

Other spellings: `FCZ`, `.ASCIZ`, `.STRZ`.

```
FCN /text/
```

Emit a string followed by a zero byte.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FCS

Other spellings: `.ASCIS`, `.STRS`.

```
FCS /text/
```

Emit a string with the high bit set in its last character.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FDB

Other spellings: `.WORD`, `.DW`.

```
FDB word[,word…]
```

Emit 16-bit words, high byte first.

LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FDBS

```
FDBS word[,word…]
```

Emit 16-bit values low byte first, opposite to the processor's normal word order.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FILL

```
FILL byte,count
```

Emit the given byte repeatedly.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

FQB

Other spellings: `.4BYTE`, `.QUAD`.

```
FQB value
```

Emit 32-bit quantities, high byte first.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFDEF

```
IFDEF symbol
```

Test whether a symbol has already been defined.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFEQ

```
IFEQ expression
```

Include the following block when the expression equals zero.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFGE

```
IFGE expression
```

Include the block when the expression is zero or positive.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFGT

```
IFGT expression
```

Include the block when the expression is positive.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFLE

```
IFLE expression
```

Include the block when the expression is zero or negative.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFLT

```
IFLT expression
```

Include the block when the expression is negative.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFNDEF

```
IFNDEF symbol
```

Test whether a symbol is still undefined.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFNE

Other spellings: `IF`.

```
IFNE expression
```

Include the block when the expression is nonzero.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFP1

```
IFP1
```

Legacy pass-number conditional. LWASM warns that it cannot reproduce pass-one assembly semantics.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFP2

```
IFP2
```

Legacy pass-number conditional. LWASM warns that it cannot reproduce pass-two assembly semantics.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFPRAGMA

Other spellings: `IFOPT`.

```
IFPRAGMA option
```

Include a block according to the current pragma setting.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

IFSTR

```
IFSTR comparison,arguments
```

Include a block according to the specified string comparison.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

INCLUDE

Other spellings: `INCL`, `LIB`, `USE`.

```
INCLUDE filename
```

Read another source file at this location.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

INCLUDEBIN

```
INCLUDEBIN filename
```

Copy bytes from a binary file into the assembly output.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

INCLUDESTR

```
INCLUDESTR string
```

Treat the contents of the given string as additional assembler source.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

MACRO

Other spellings: `MACR`.

```
name MACRO
```

Begin defining a sequence of source lines for later expansion.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ORG

```
ORG address
```

Set the address of the following output. In DECB an origin determines where a data block loads.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

PHASE

```
PHASE expression
```

Give following code a different logical address for labels and expressions while retaining its output placement.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

PRAGMA

Other spellings: `*PRAGMA`, `**PRAGMA`, `OPT`.

```
PRAGMA option
```

Change a LWASM dialect option. A target-changing pragma affects subsequent assembly.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

REORG

```
REORG
```

Restore the location associated with the previous ORG.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

RMB

Other spellings: `.BLKB`, `.DS`, `.RS`.

```
RMB count
```

Advance the location counter by a number of bytes. Use ZMB when the load image must contain zeros.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

RMD

Other spellings: `RMW`.

```
RMD count
```

Reserve 16-bit words.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

RMQ

```
RMQ count
```

Reserve 32-bit values.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

SECTION

Other spellings: `SECT`, `.AREA`.

```
SECTION name
```

Begin a section in an object module. The linker determines final placement.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

SET

```
name SET expression
```

Assign a symbol a value that later SET statements may change.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

SETDP

```
SETDP page
```

Tell the assembler the assumed direct page. Guest software must establish the matching DP value.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

SETSTR

```
SETSTR name=string
```

Set an assembler string variable.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

STRUCT

```
name STRUCT
```

Begin a structure layout. Field labels describe offsets in its instances.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

WARNING

Other spellings: `MSG`.

```
WARNING message
```

Report a warning with the supplied text.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ZMB

Other spellings: `BSZ`, `FZB`.

```
ZMB count
```

Emit the requested number of zero bytes.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ZMD

```
ZMD count
```

Emit zero-filled words.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

ZMQ

```
ZMQ count
```

Emit zero-filled four-byte quantities.
LWASM directive. Its effects take place during assembly.
Reading: Astle, LWTools 4.25, ch. 3.

Dialect, expansion and undocumented spellings

Check the listing before counting debugger steps. A spelling can denote an actual HD6309 operation in one build and a sequence emitted by an assembler pragma in another. The names grouped below describe separate assembler features. The instruction directory identifies the corresponding hardware operations.

ASM-09 dialect

Spellings in this group: `IFC`, `IFNC`, `EXITM`, `EXIT`.
The ASM-09 compatibility dialect adds string equality and inequality conditionals and an early macro exit.
Reading: Astle, LWTools 4.25, ch. 3.

Assembler conveniences

Spellings in this group: `ASRQ`, `CLRQ`, `COMQ`, `LSLE`, `LSLF`, `LSLQ`, `LSRQ`, `NEGE`, `NEGF`, `NEGW`, `NEGQ`, `TSTQ`.

LWASM can expand convenience operations into several instructions when the corresponding pragma is active. The listing gives the actual expansion. A 6809 emulation of CLRD, for example, executes CLRA and CLRB separately.

6809 expansions include ASLD, ASRD, CLRD, COMD, LSLD, LSRD, NEGD and TSTD. These names also identify real HD6309 operations. Additional 6309 convenience spellings include ASRQ, CLRQ, COMQ, LSLE, LSLF, LSLQ, LSRQ, NEGE, NEGF, NEGW, NEGQ and TSTQ. MC6800 compatibility is a separate optional dialect.

Reading: Astle, LWTools 4.25, ch. 3.

Compatibility directives

Spellings in this group: `NAM`, `PAG`, `PAGE`, `SPC`, `TTL`, `.BANK`, `.MODULE`.

LWASM accepts these listing and tool-compatibility directives without generating code.

Reading: Astle, LWTools 4.25, ch. 3.

MC6800 compatibility

Spellings in this group: `ABA`, `CBA`, `CLC`, `CLF`, `CLI`, `CLIF`, `CLV`, `CPX`, `DES`, `DEX`, `DEY`, `INS`, `INX`, `INY`, `SBA`, `SEC`, `SEF`, `SEI`, `SEIF`, `SEV`, `TAB`, `TAP`, `TBA`, `TPA`, `TSX`, `TXS`, `WAI`.

These optional LWASM spellings expand to 6809 encodings with the 6800compat pragma. Expansion can require several instructions. Inspect the listing when stepping.

Reading: Astle, LWTools 4.25, ch. 3.

OS-9 output directives

Spellings in this group: `OS9`, `MOD`, `EMOD`.

OS9 encodes SWI2 followed by a service byte for OS-9 or object output. MOD and EMOD frame an OS-9 module. Assembly Development runs standalone Disk BASIC block images.

Reading: Astle, LWTools 4.25, ch. 3.

Undocumented and emulator opcodes

Spellings in this group: `HCF`, `RHF`, `RESET`, `HALT`, `BREAK`, `LOG`.

LWASM lists HCF, RHF and RESET for certain 6809 encodings. Behavior outside the documented instruction set can differ by chip. BREAK and LOG are emulator extensions. VCC Plus's Mac decoder also excludes the optional HALT/BREAK encodings.

Reading: Atkinson (2009), p. 153 · Astle, LWTools 4.25, ch. 3 · VCC Plus, Native Debugger, disassembly.

Keep a workbench notebook

Appendix C

Keep a record short enough that you will actually use it. For a surprising byte, write down the guest CPU and mode, the source label, the PC of the *next* instruction, the address space and the byte you expected. If you share only a hexadecimal address, another reader can inspect precisely the wrong thing.

Read the original assembler manual

The complete LWTools 4.25 manual is available offline alongside this book, with its original source, PDF and license notices. Consult it for LWASM syntax and LWLINK scripts; the bundled command-line help also records installed options and VCC Plus extensions. **Ask the Manuals** and project MCP manual tools search these references together with the teaching books. The complete CMOC manual is useful when linking assembly with C.

Save the examples

Question to test	File to save	Setup and destination
Did the message survive?	dispatch.asm	Either CPU, Disk BASIC. Change the last letter in a copy.
Which physical block backs this window?	bankwalk.asm	Use the disposable setup in chapter 4.
Which page supplied the pixels?	PAGES.bas	Open in BASIC Development and use the two stops in chapter 5.
What does W count?	dispatch6309.asm	HD6309 guest and target. Stop before and after TFM.
Which mode did the program leave behind?	mode.asm	Fresh HD6309 Disk BASIC, MD=$00 before starting.
How does a software interrupt return?	softirq.asm	Follow the setup and restore sequence in chapter 7.
Can I rebuild and deliver two modules?	Dispatch project	Expand the ZIP into a writable folder, then open the project folder as described in chapter 8.

In the app, selecting an assembly file or project archive opens a save panel. Work on the copy you saved. A BASIC example link opens its own BASIC workspace. In the browser edition you can download the files. With an unmodified checkout the whole shelf also opens locally from **docs/manuals/index.html**.

Choose the right address book

A 6809 address runs from $0000 to $FFFF. It answers “where in the guest CPU’s current view?” For a RAM location, the enabled MMU, active task and block select the backing page. The physical-video base answers a different question about what the video hardware displays. A ROM overlay can change what the CPU reads in part of that space. Addresses near the top can reach device registers and vectors; a read or write performed by guest code may have a hardware side effect.

- To inspect a label in your assembled listing, begin with **Mapped CPU**. Look up its backing using **MMU** before comparing it to a physical address.
- To ask why a screen didn’t change, inspect **Current Video Page** as well as the chosen address. Record the graphics mode, palette set and whether the address is mapped or physical.

- To call a BASIC service, identify its arguments, registers you need afterward and how you will return. The exercises assume the Disk BASIC environment. CHROUT uses a vector at $A002 and accepts the character in A. The planet program also uses keyboard polling. Read the key-to-color walkthrough for a concrete call site.
- To change a mapping or install an interrupt handler, first save any work you care about. Keep the executing code and its stack reachable, save the state you will modify, and plan a path that restores it. Write down a checkpoint *after* restoration.

Get a known BASIC prompt

If you stepped inside a ROM service, place a stop at the next instruction in your own source and **Continue**. If you stopped just before the final RTS of a direct-run exercise, continue through the return. At the BASIC prompt type **PRINT 2+3**. A fresh answer of 5 distinguishes an interactive return from an old OK still on screen.

An example that ends in a deliberate hold loop keeps executing the loop. Save the source; restart the disposable guest to move on. A program stopped partway through an MMU, mode or vector change still has that intermediate state. Follow the documented cleanup path if it is reachable; otherwise restart the practice setup. Avoid changing the PC to a convenient-looking address while the stack still belongs to an interrupt handler.

Books and tools behind the experiments

- Motorola, Inc. *MC6809–MC6809E 8-Bit Microprocessor Programming Manual*. M6809PM (AD), March 1981, reprinted May 1983. Chapters 1–2 and Appendix A. Registers, addressing, condition codes and the interrupt frame.
- Tepolt, Laurence A. *Assembly Language Programming for the CoCo 3*. TEPCO, 1987. Figures 3-4 and 4-1, pp. 20 and 23. Historical context for the relationship between processor and video addressing in the CoCo 3.
- Atkinson, Darren. *Motorola 6809 and Hitachi 6309 Programmer's Reference*. 2009. Parts I and II. Comparative instruction, addressing and mode descriptions.
- Burke, Chris. *The 6309 Book: Inside the 6309 Microprocessor*. Second edition. Burke & Burke, 1993. Extended registers and native-mode behavior.
- Barden, William, Jr. *TRS-80 Color Computer Assembly Language Programming*. Radio Shack, 1983. Subroutines, addressing and machine-language practice on the Color Computer.
- Astle, William, and LWTools contributors. *LW Tool Chain*. Manual distributed with LWTools 4.25. LWASM and LWLINK chapters. Source syntax, object sections, linking and output formats.

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

Original VCC Plus prose and instructional images © 2026 Brian K. Holman. The Hitachi badge on the cover identifies the processor. 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

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`.

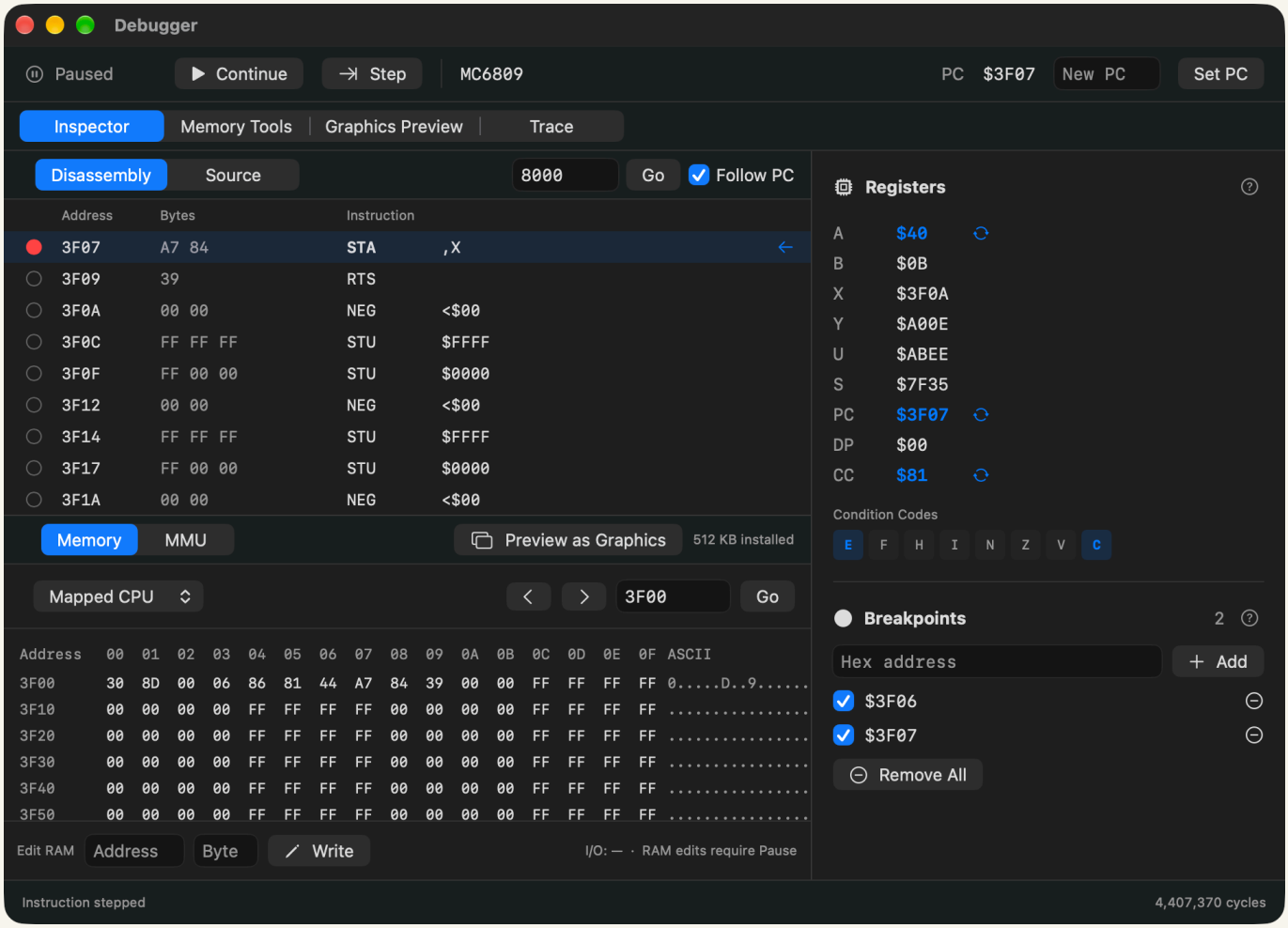


Figure 1: The paused Inspector points at STA at $3F07. The register pane shows A=$40 and X=$3F0A. The value in column 0A of row 3F00 is 00.

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

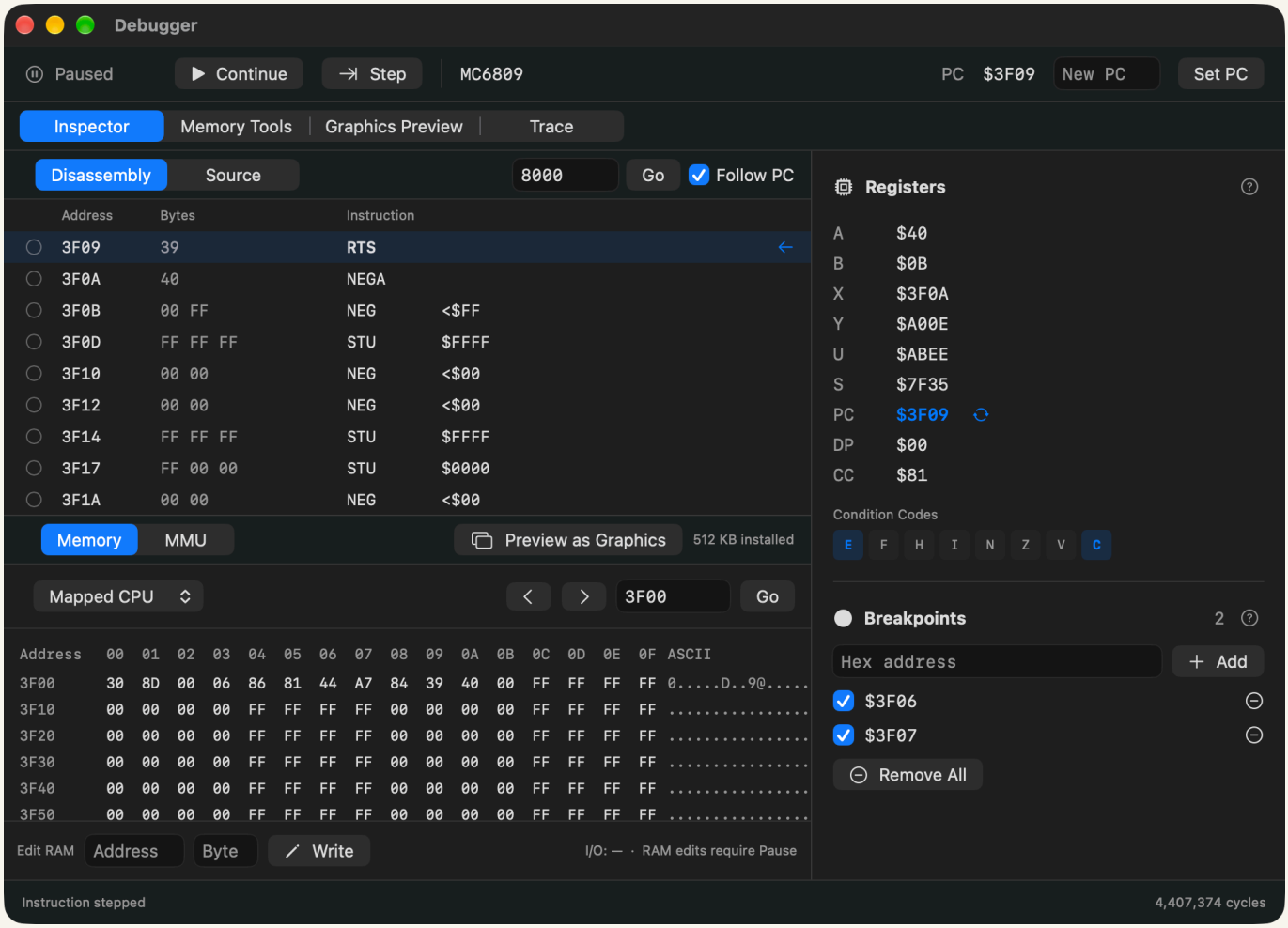


Figure 2: After one machine step the instruction arrow points to RTS at $3F09. Row 3F00, column 0A now contains 40. X still points to $3F0A.

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

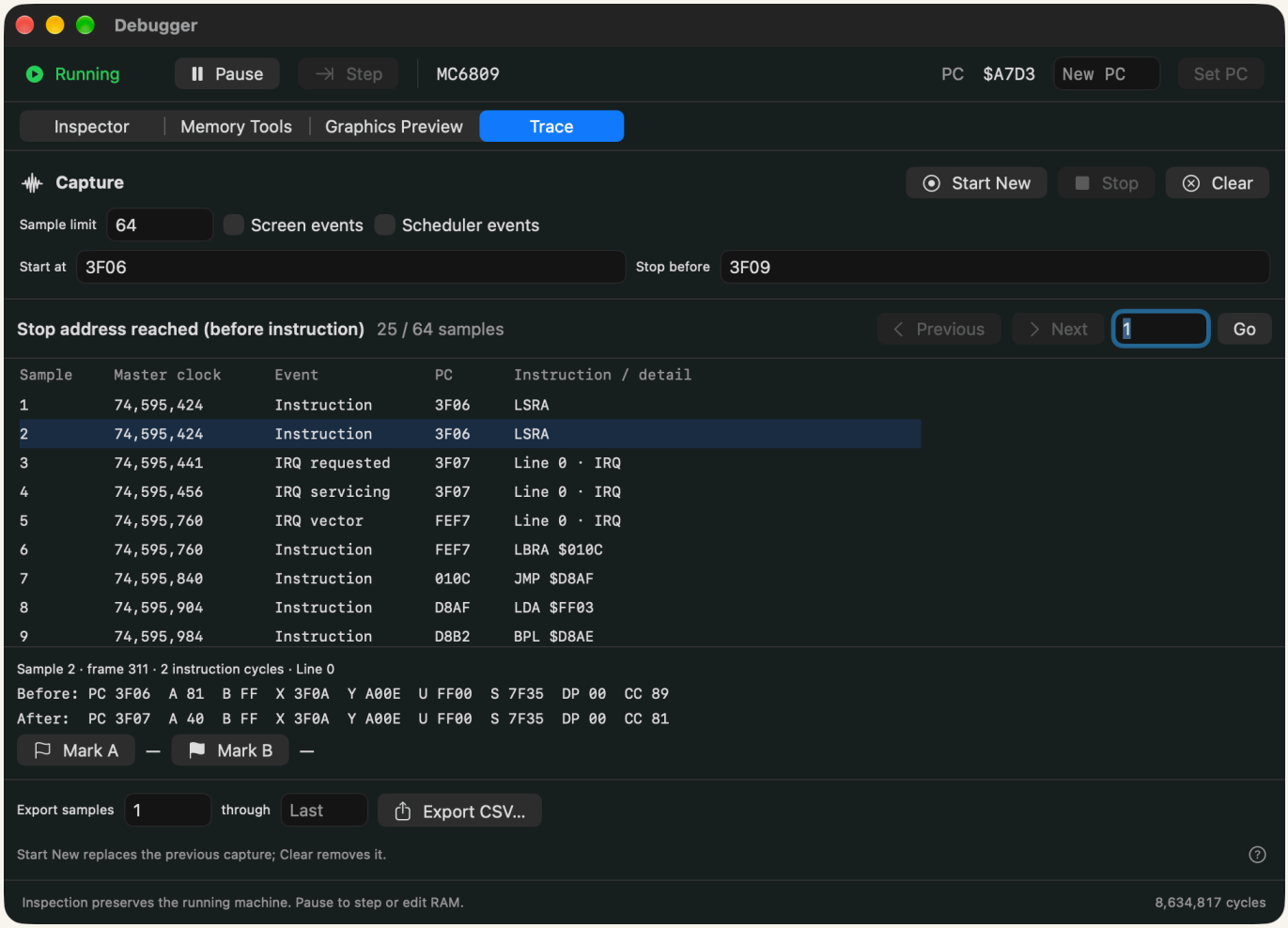


Figure 3: The trace is complete and the emulator is running. The selected LSRA sample shows two instruction cycles, with PC changing from 3F06 to 3F07 and A from 81 to 40.

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.

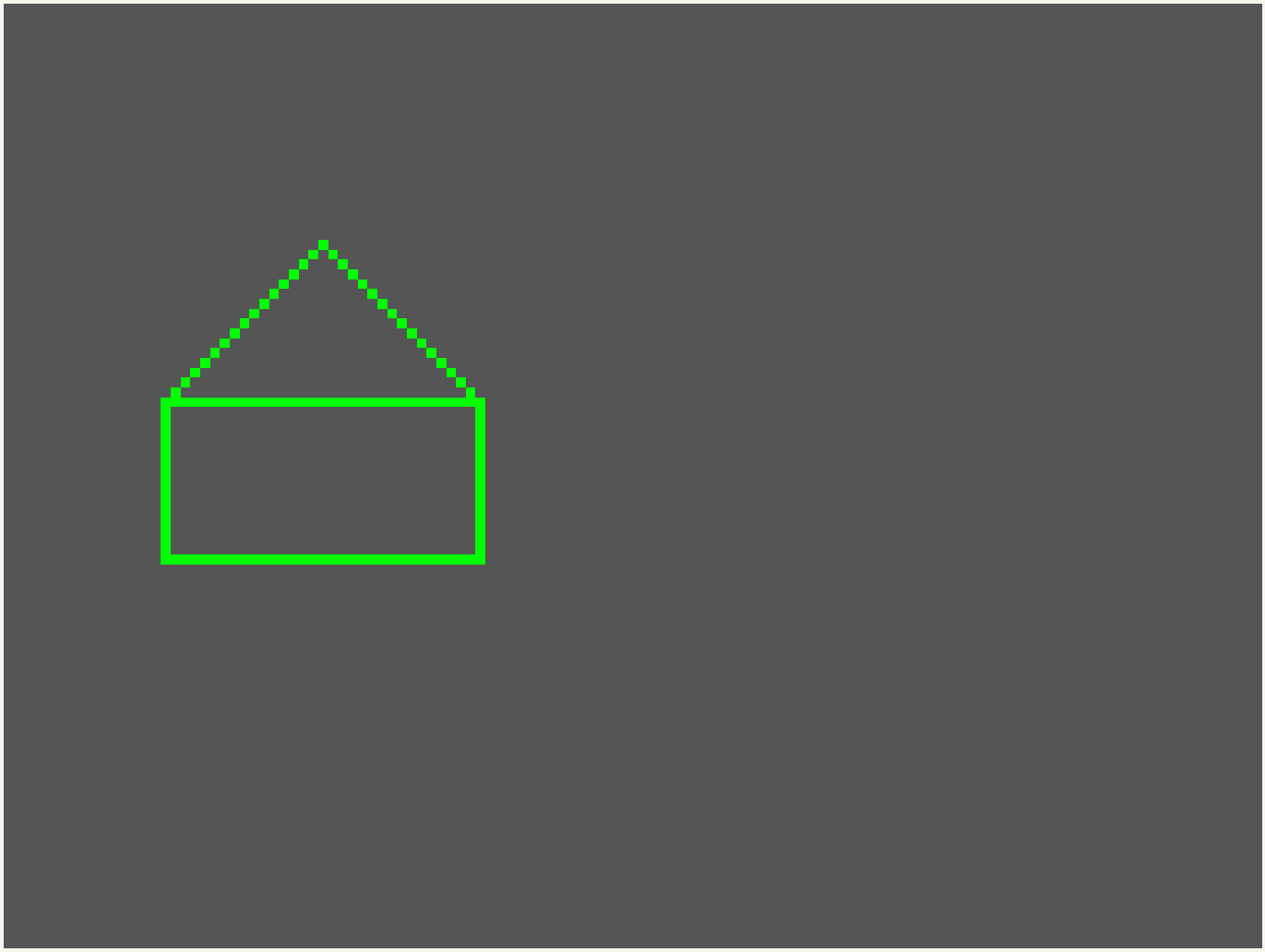


Figure 4: Current Video Page at line 80. A green house stands at the left.

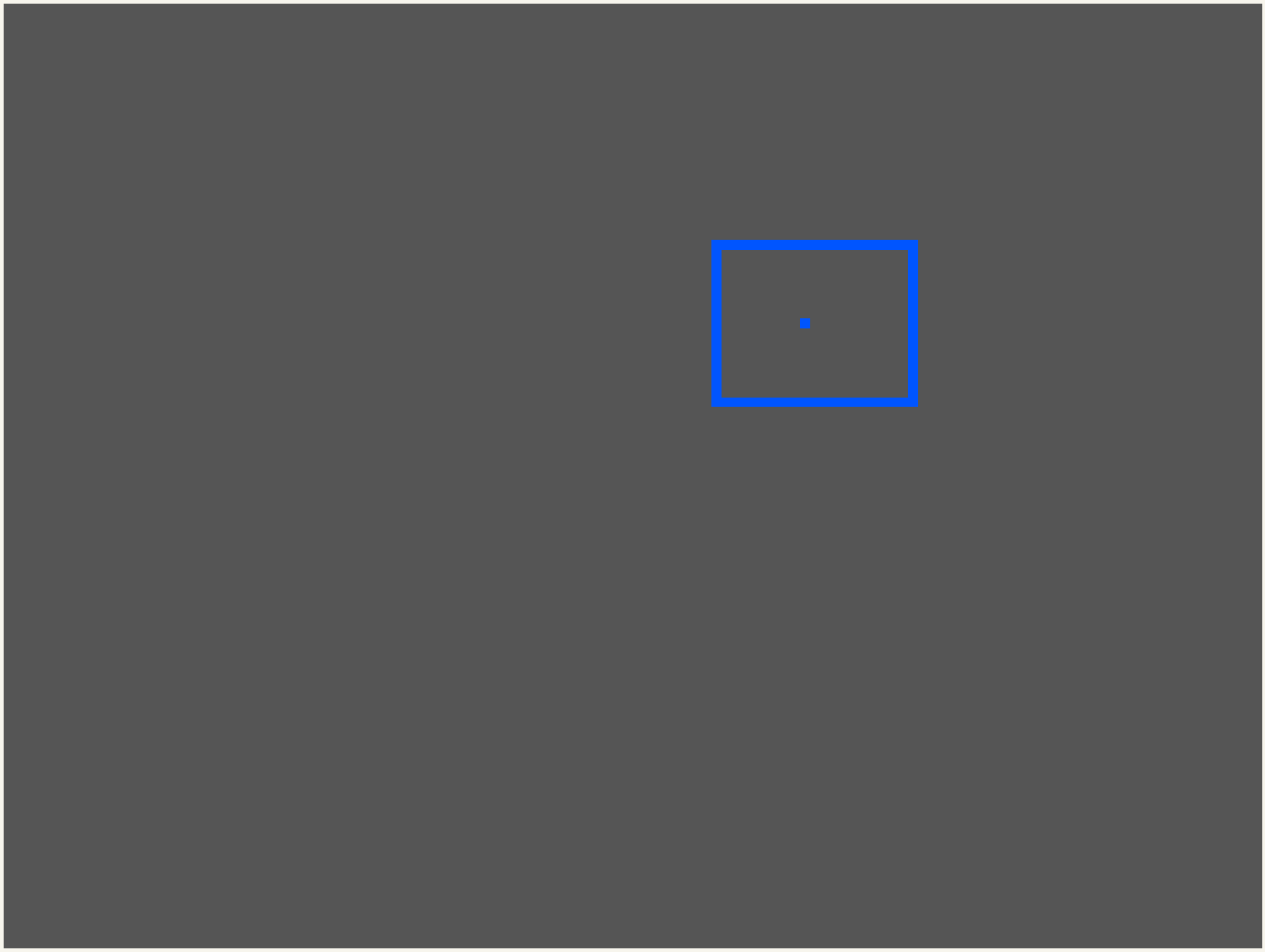


Figure 5: At Address at the same stop. A blue rectangular receiver at the right contains one blue dot.

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**.

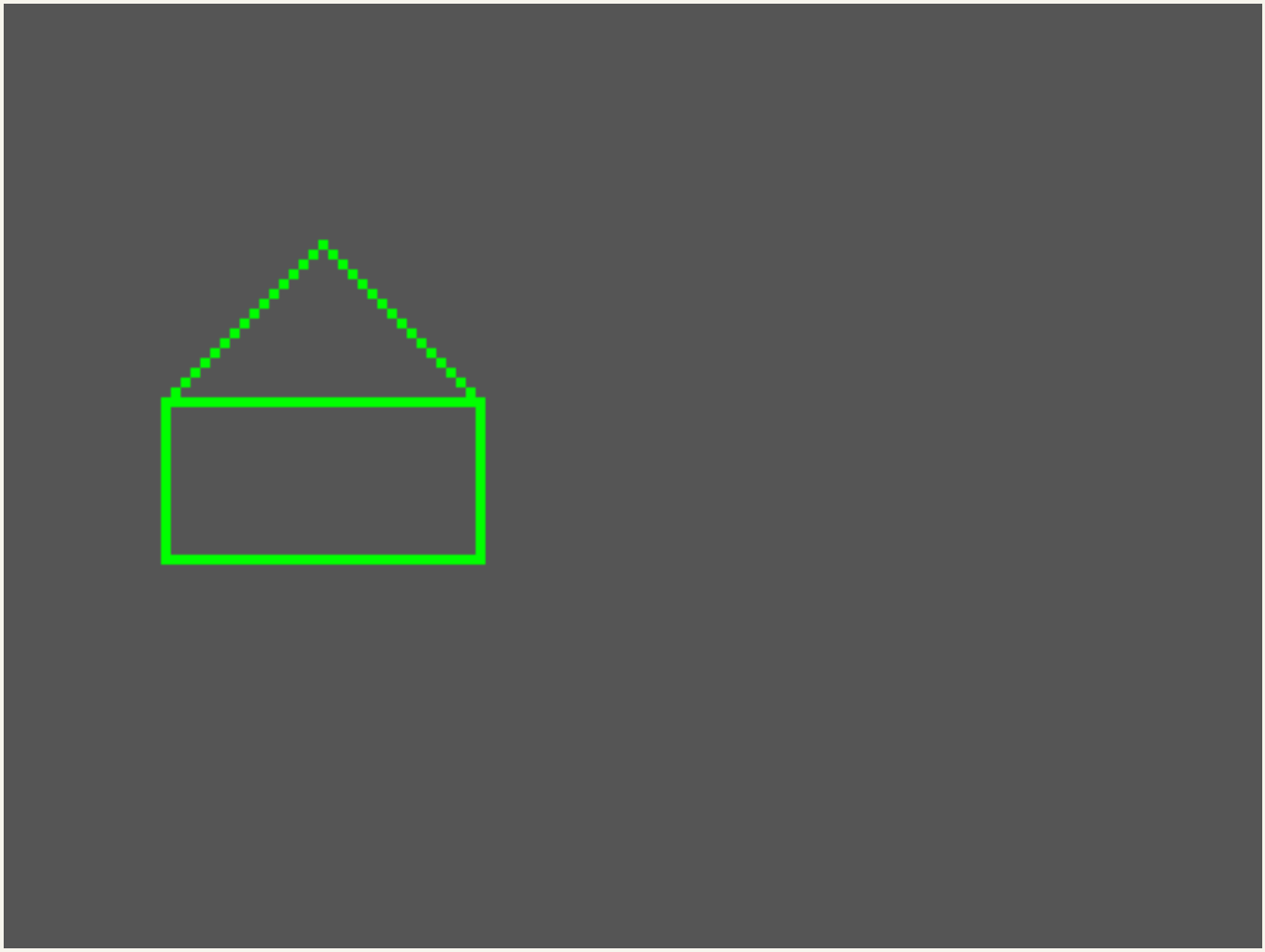


Figure 6: CoCo monitor before the first tap. A green house appears at the left on a gray background.

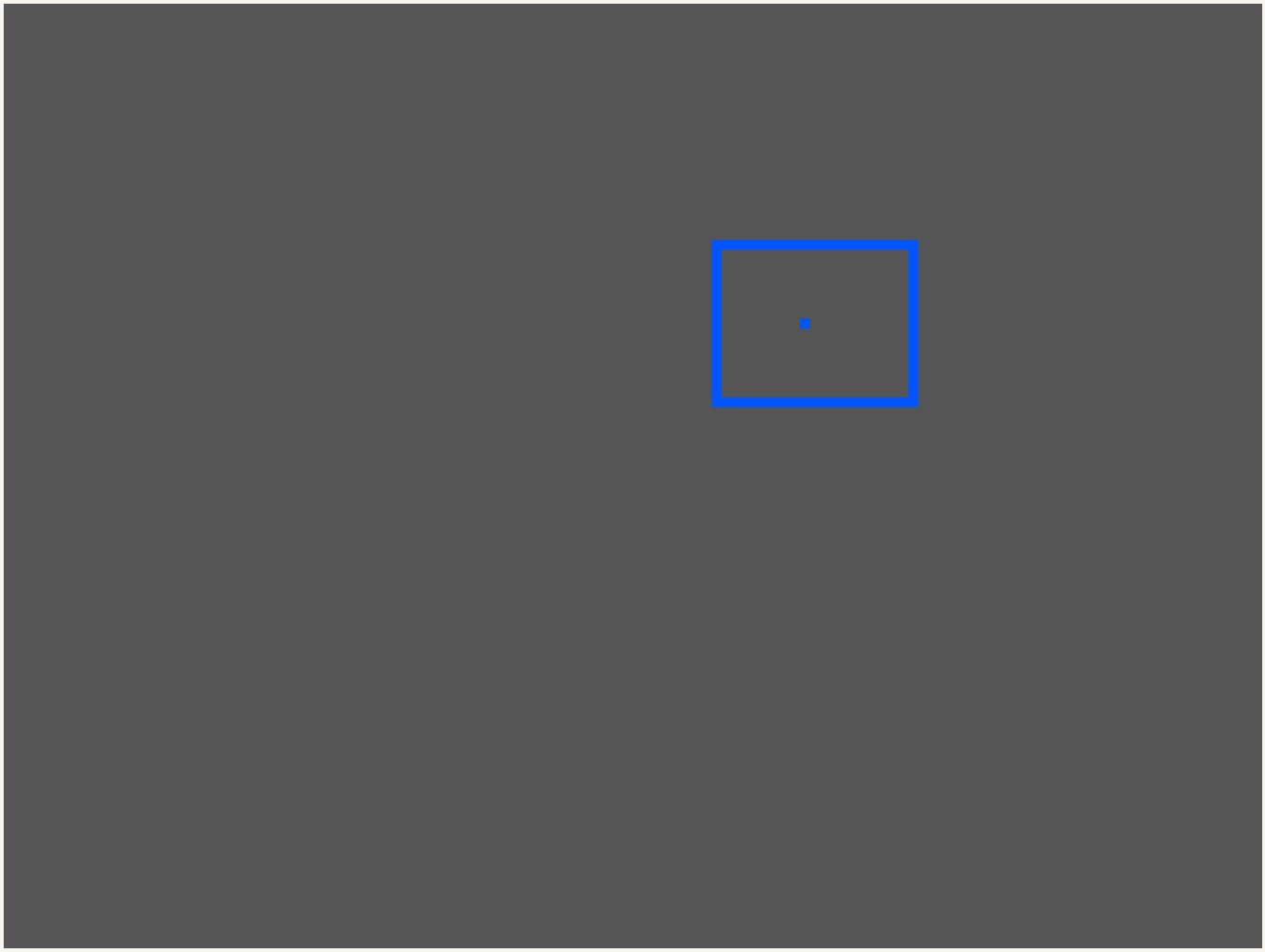


Figure 7: CoCo monitor after the first tap. A blue receiver at the right encloses the restored dot.

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.

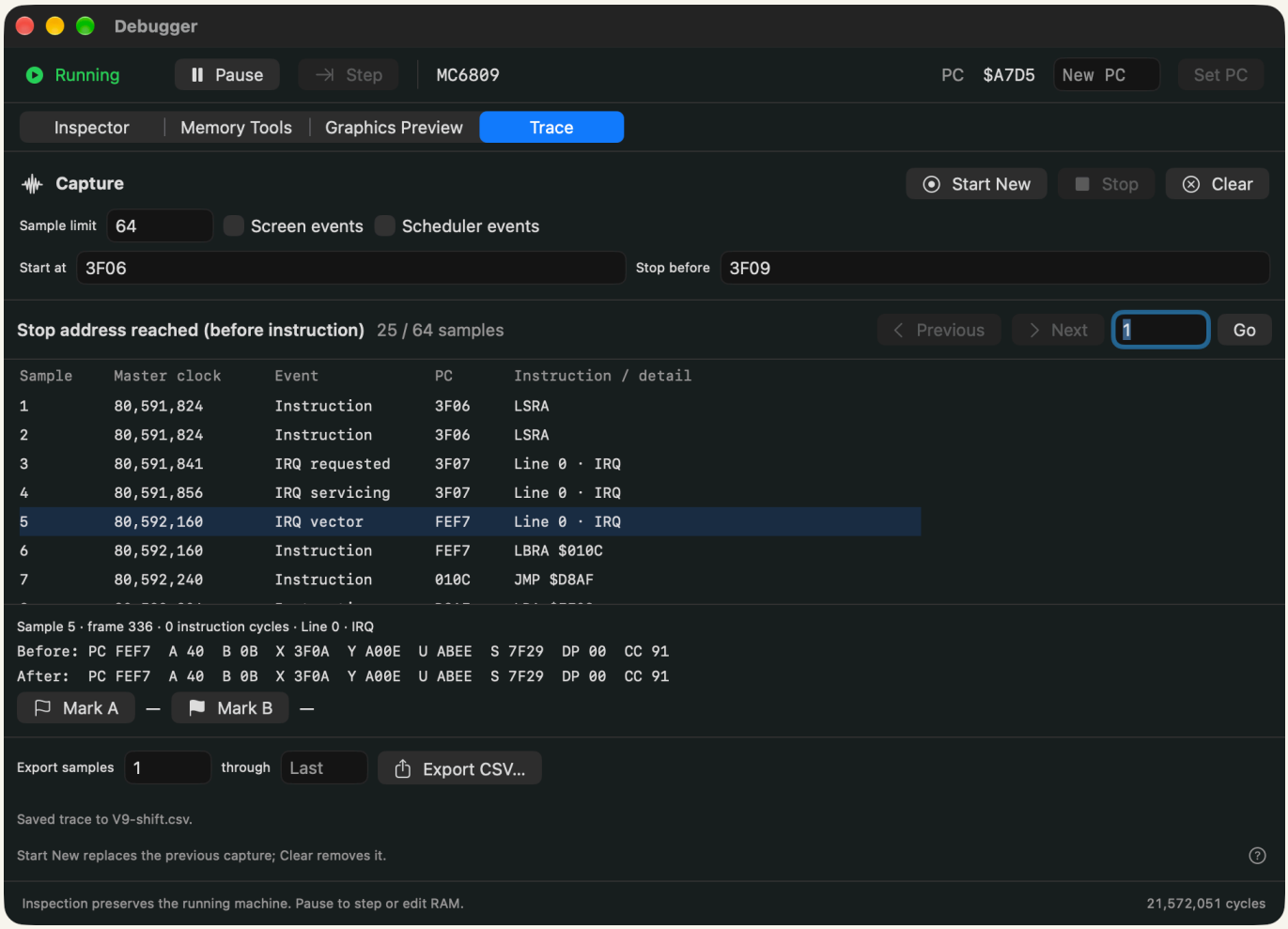


Figure 8: Native Trace shows IRQ requested, IRQ servicing and IRQ vector before the handler's first instruction at FEF7.

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.

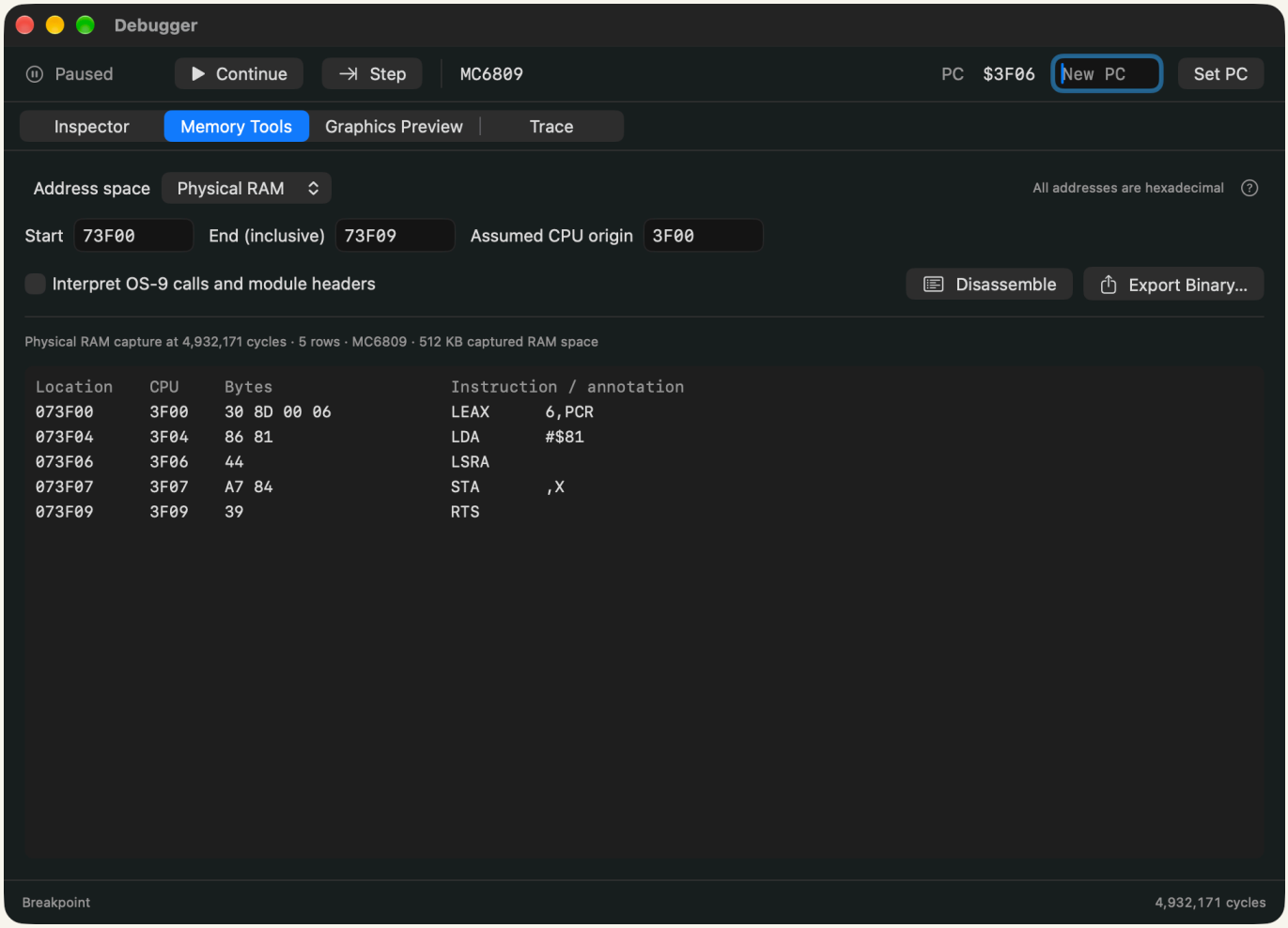


Figure 9: Memory Tools decodes five instructions from physical 073F00 through 073F09, with CPU origin 3F00. Export Binary is available above the rows.

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

Working with OS-9 — A Comprehensive Manual

Build a beacon station on your CoCo. Follow a report from a shell command through a process, a memory map and a device driver. Then write the program that sends it.



The running CoCo shows OS-9 assembling and identifying the station's beacon module.

Brian K. Holman

Book 11 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Which program heard me? 3
Give the shell a job 5
Where did the output go? 7
A file has a name and an owner 9
Follow a filename onto the disk 12
Recover the right copy 15
Catch a process while it sleeps 17
Where does a program live? 19
One address, two views 21
Open a device by name 24
Who owns this terminal? 26
Put the night watch on duty 28
Assemble a program on the CoCo 31
Deliver the beacon report 34
Find a command 36
Device and storage atlas 41
System calls and errors 43
Choose a language and a delivery format 46
Find your way back into the system 49
Publication and artwork notices 51

Which program heard me?

Chapter 1 of 14

Your station needs a log. At the keyboard you can name a disk, ask the time, or start a second command interpreter. The first puzzle is who hears those keystrokes. On this CoCo the answer changes twice between power-on and the first command.

Boot your practice station

In **Machine → Configuration Profiles…**, select **NitrOS-9 L2 V3.3.0**, choose **Create Fresh Starter Media…** and name the personal copy **Beacon Station**. Load that copy, confirm the restart and choose **Run**. Use the same working images for this book. If you are returning from the quick start, you can also use its personal practice set if the new names are unused. The quick start shows the controls.

The prepared profile enters the boot command at RGB-DOS. The CoCo boots the system on HD0 and eventually starts **Shell+** at the terminal. Click the CoCo display. Wait for the prompt ending in **/DD:**. The text before the path identifies the terminal and shell process, such as **{Term|02}**. If you see the BASIC **OK** prompt, follow the return trip to OS-9.

Open **Media → Virtual Hard Disks…**. Identify the image files assigned to HD0 and HD1. The guest uses **/dd** for its default system disk. On this profile **/h0** reaches that same disk. **/h1** holds your practice work. Close the drive window and click the CoCo display again.

Read the system's morning instructions

At Shell+ enter:

```
list /dd/startup
```

You are reading a file already run during boot. Near the top, two echo commands explain the welcome banner. The file keeps the shell resident, loads a set of utilities, and loads a networking module. Farther down it creates windows and starts shells on window devices. Seeing a networking module in this listing does not grant a guest permission to use a Mac shell. That decision belongs to VCC Plus.

The kernel and I/O modules must already be running for that file to be readable. The boot code brought in the operating system, including the modules in **OS9Boot**. The system startup process arranges the initial environment. The shell can then execute the readable startup procedure. Changing a startup command and rebuilding the set of boot modules act at different stages. Keep the system image intact during these exercises. Later chapters let you inspect modules while the station runs.

Predict two search paths

Make a new practice directory on your private HD1 and enter it.

```
makdir /h1/station
chd /h1/station
pwd
pxd
```

pwd returned **/H1/station**, the current data directory. **pxd** returned **/DD/CMDS**, the current execution directory. If the directory already exists, inspect its contents before reusing it. A new copy of the practice media gives you fresh names.

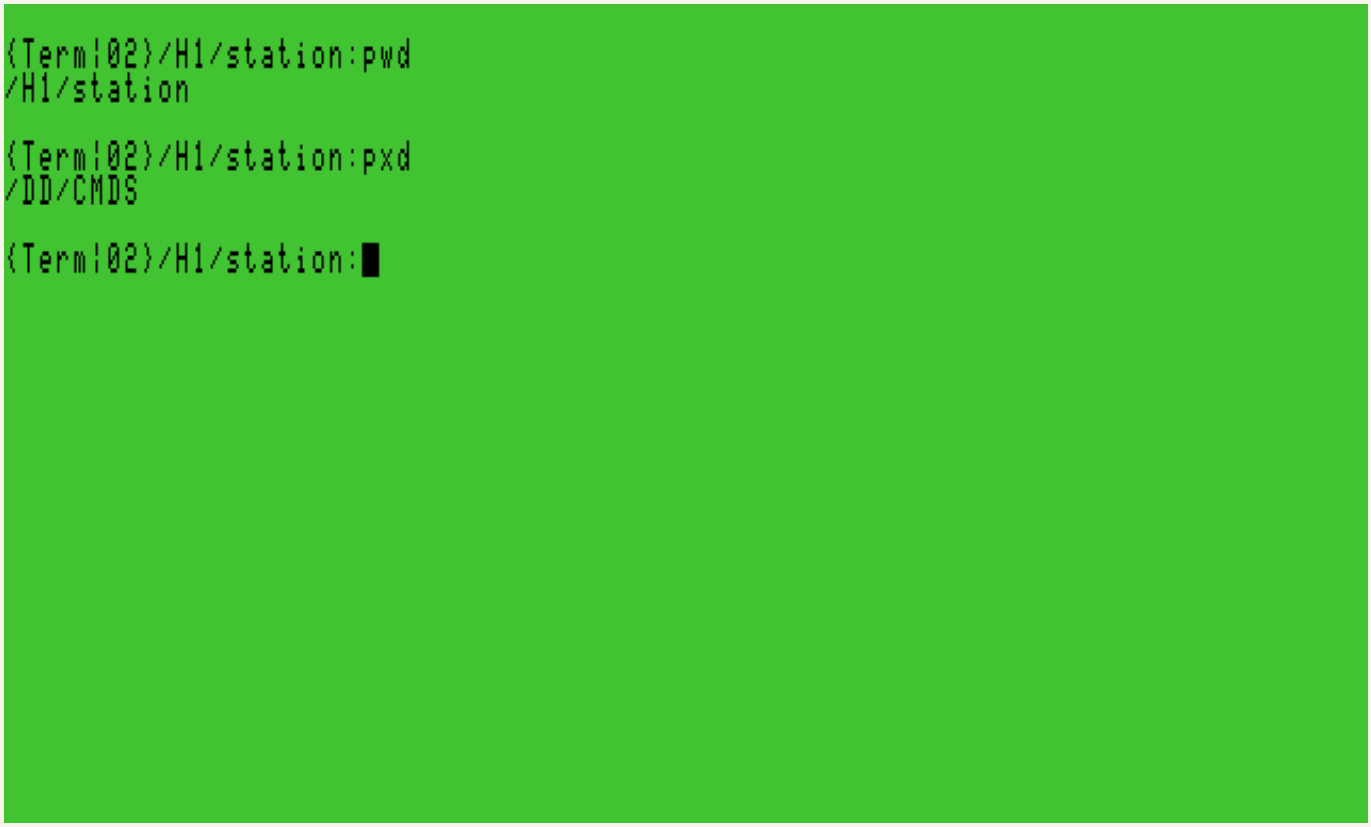


Figure 1: The guest prints /H1/station for pwd and /DD/CMDS for pxd beneath the same Shell+ prompt.

When you ask a utility to open an ordinary relative filename, the data directory supplies the starting point. When you type a program name, Shell+ can find a resident module or search for executable material using its command search rules. The execution directory supplies the usual location of disk commands. A leading slash starts a path at a named device. You can therefore stand in your station directory, run **list** from the command environment and read **/dd/startup** from the system disk.

Prediction. You will shortly type **build report**. Which disk will receive the two-line file? The answer is HD1 because **report** is a relative data filename and the current data directory is **/H1/station**. The shell's ability to find **build** is a different lookup.

Give the shell a job

Chapter 2 of 14

A station operator can remember two commands. A relief operator arriving tomorrow would appreciate finding them already written down. Give the next shift a file that the shell itself can read.

Write and inspect a procedure

Start in **/H1/station**, the data directory created in chapter 1. Enter **build report**. The **?** belongs to Build. Type each of the following lines, pressing Return after each. At the next empty **?** press Return once more. That blank reply ends entry.

```
echo beacon online
pwd
```

Now enter the following at Shell+:

```
list report
shell report
```

list displays the two command lines exactly as stored. **shell report** interprets those lines and displays **beacon online** followed by **/H1/station**. The first word on the outer command line names a program. The word **report** is an argument to it. Inside the file, **echo** receives the words **beacon online**. The second command, **pwd**, uses the directory inherited from the calling environment.

Notice what the procedure leaves behind. You are back at the prompt that started it, still standing in the station directory. A child shell can change its own working directory while running a job. Its parent retains its own directory when the child exits.

See which punctuation the shell handles

Enter these two lines separately:

```
(echo north; echo south)
echo "two words"
```

The first prints **north**, then **south**. The semicolon separates commands that run in sequence. The parentheses group that sequence. The second line prints **"two words"**, complete with quotes. The quotes survive in the parameter text seen by this version of Echo. This matters to a reader coming from a Mac shell, where the output of the same-looking command can differ.

At the OS level, the new program receives a parameter area containing text. Each utility has rules for interpreting that text. Consider the different jobs of **shell report**, **list report** and **echo report**. The same word can name a procedure, name a file to display, or simply be part of a message. Use each utility’s own syntax when an argument can contain a space.

Let the child read until the file ends

Try another way to supply those two lines:

```
shell <report
```

This time an interactive copy of Shell+ receives the file on its standard input. In the practice session it printed its greeting, ran both commands, displayed **eof**, and returned to the original shell. The terminal and process number in the child's prompt help identify the change. The child stops because the file has no more input. Nothing in the file needs to type an exit command.

Programs also return a completion status. A child can finish successfully or give its caller an error number. Output text and completion status are separate ways to report what happened. In the next chapter a failing child's message will travel down a path you choose. Later you will write a module that chooses its own exit status.

Where did the output go?

Chapter 3 of 14

Two operators want different things from the same report. One wants to see a single matching line. The other wants both a display and a saved copy. Neither should have to rewrite the program producing the text.

Connect paths zero, one and two

A path number is a small handle valid in the process using it. A newly started command ordinarily inherits three paths from its parent. At an interactive shell, more than one of these can reach the same terminal.

Path	Conventional use	Change it in Shell+
0, standard input	Read a key or the next bytes of an input stream	**<** followed by an input filename
1, standard output	Deliver the command’s regular result	**>** followed by a new output filename
2, standard error	Deliver diagnostic messages, status and prompts	**>>** followed by a new diagnostic filename

Shell+ arranges the destinations before starting the child. A program that writes to path 1 naturally follows the new destination. A program that opens **/Term** explicitly has made a different choice. In particular, **>>** on this shell selects path 2. Treating it as a request to append to an existing report can produce an error or put the wrong stream in a file.

Filter a listing, then split a stream

From **/H1/station** run these commands. Choose an unused name for **archive** if repeating the exercise.

```
list report ! grep beacon
echo radio ! tee archive
list archive
```

The exclamation mark joins one command’s output to another command’s input. In the first pipeline, List sends both lines of the procedure down a pipe. Grep emits the line containing **echo beacon online**. The **pwd** line fails the match. In the second pipeline, Tee reads the line produced by Echo, writes it to the file **archive**, and also sends it to its own standard output. You see **radio** twice, once during the pipeline and once when List reads the saved file.



Figure 2: The running guest filters the report procedure, displays radio from a pipeline and reads radio back from the file named archive.

A pipe is a stream between open paths. OS-9 can schedule the reader while the writer is producing more bytes. Once every writer closes its path and the reader consumes the remaining bytes, the reader reaches end of input. A disk file has a directory entry and remains available to a future process. You can come back to **archive** after a reboot.

Catch a diagnostic separately

Use **build fail** in your practice directory. Give it one line, **list absent**, and then an empty Return. Leave **absent** as a deliberately unused filename. At Shell+ type:

```
shell <fail >>errors
list errors
```

The child shell tries to list a missing file. Most of the usual terminal chatter disappears. Opening **errors** afterward reveals its prompts, **ERROR #216**, and **eof**. In our session a short Shell+ banner still appeared on the terminal. This tells you why a useful diagnostic capture includes path 2 even when some messages still reach path 1. The file **errors** needs a fresh name for a second run.

The earlier filter, file copy and diagnostic capture all pass text through the same three numbered paths. Remember path 1. The report writer you assemble at the end of the book uses that number directly.

A file has a name and an owner

Chapter 4 of 14

Before changing a procedure, you want a known good copy. A name helps you find it. To decide whether you can read, change or run it, OS-9 consults information stored in its file descriptor.

Compare a working file and a copy

Begin in **/H1/station** with the original **report**. Use an unused destination name.

```
copy report backup
cmp report backup
attr backup
dir -e /h1/station
```

Cmp compared **00000017** bytes and found **00000000** differences. Those counts are hexadecimal. The two files each hold 23 bytes. A comparison of contents answers a narrower question than a directory listing. It says the bytes match. The detailed listing shows the files have distinct descriptor sectors and gives their owner, modification timestamp, attributes and size.

The attributes of **backup** read **------wr**. Read the positions from right to left: owner read, write and execute, followed by public read, write and execute, the share-mode bit, and the directory flag. Here only the rightmost two bits are set. When OS-9 checks a file, it considers the user ID of the caller as well as the access requested. A permission to write an existing file and a permission to create entries in a directory answer different questions.

Give the backup a durable name

```
rename backup report.old
shell report.old
echo new >report
list report
```

Rename changes the directory entry. The saved copy still runs and prints **beacon online** and your working directory. The attempted create over **report** returns error 218. The original procedure still lists both commands. On this shell a fresh filename is the simple way to keep a second run’s output.

```
 Directory of /h1/station  2026/11/06 19:27

Owner  Last modified   Attributes Sector Bytecount Name
----- ---------------- ---------- ------ --------- ----
   0  2026/11/06 19:27  ------wr      28        17 report
   0  2026/11/06 19:27  ----r-wr      30         7 archive
   0  2026/11/06 19:27  ------wr      38        17 backup


(Term!02)/H1/station:rename backup report.old

(Term!02)/H1/station:shell report.old
beacon online
/H1/station

(Term!02)/H1/station:echo new >report
ERROR #218

(Term!02)/H1/station:list report
echo beacon online
pwd

(Term!02)/H1/station:
```

Figure 3: Shell+ shows the renamed procedure running, refuses to create over report, and lists both original lines afterward.

Diagnose a permission failure

Use a separate copy so you can observe a real access error and repair it without changing either copy of the lesson procedure. The following assumes **guard** is unused.

```
copy report guard
attr guard -r
list guard
attr guard r
list guard
```

Removing the owner read bit produced **------w-**. The first List returned **ERROR #214**. Restoring the bit produced **------wr**. The last List recovered the original two lines. The name and bytes survived throughout. This is a different problem from error 216, where the requested pathname could not be found.

For a program module, owner execute is a separate attribute. Chapter 13 sets that bit on the station's own assembled program. The right to execute a file and the operating system's ability to recognize its module header are both relevant there.

Inspect a supported image without booting it

For a supported raw OS-9 image, **Edit...** beside its mounted floppy or hard drive provides a native directory browser. For a separate unmounted image, use **Media → Floppy Disks... → Open Disk Image...**. If the image contains more than one filesystem, select its OS-9 volume before browsing the folders. Select a file or folder and choose **Info**. The same File Information sheet shows its details, owner/public read-write-execute bits, **Single access (non-shareable)** and ownership. File versus Directory is structural information and stays read-only. Single access permits only one opener at a time.

OS-9 stores one 16-bit owner ID. The editor's Group ID and User ID fields are its high and low bytes, each from 0 through 255; the table displays **group.user**. Group 12 and user 34 therefore mean decimal owner ID 3106. This is not a Unix owner/group/other permission layout: access has owner and public triplets.

Save Changes in File Information changes the editor's private working image. Undo/Redo can reverse it before the image itself is saved or applied. For a mounted floppy or hard disk, the editor pauses and locks the CoCo and requires an explicit restart before applying, so OS-9 cannot reuse an old descriptor or allocation cache. Image write protection still applies even when the host editor can repair an entry with no guest access permissions.

For regular files within that supported image, **Move To…** in the editor's toolbar or context menu chooses another OS-9 folder. Dragging selected files onto a folder uses the same operation. Permissions and ownership remain attached to the file; Undo restores the previous locations. A duplicate destination name is refused without overwriting, and the move remains private until the image is saved or applied. See Move files between OS-9 folders.

The editor identifies filesystems within supported raw hard-drive images, including the OS-9 region and numbered HDB-DOS Disk BASIC volumes when both share one image. Space totals describe the selected filesystem; saving an edited copy keeps the complete image and preserves its other regions. It does not format, repair or convert a disk. Follow the Disk Image Editor procedure and limits. The supplied station exercises continue to use guest commands, so you can observe OS-9's own file handling and permissions.

Follow a filename onto the disk

Chapter 5 of 14

What did OS-9 actually do when you asked for **/h1/station/report**? The text occupies 23 bytes. To reach those bytes, the operating system first had to read two directories and a file descriptor. You can reconstruct its route from a real working disk without writing a single sector.

Compare the file and the volume

In the station directory, enter:

```
free /h1
dir –e /h1/station
dump report
```

Our fresh practice filesystem identified itself as **VCC Shared Data**. Its capacity was **368,640 sectors** in clusters of eight. A sector here contains 256 bytes. A cluster therefore accounts for 2,048 bytes of allocation. Your free-sector count will change as files and directories appear. These figures describe the OS-9 filesystem inside the shared image. Other parts of that same image can hold BASIC volumes.

Dump starts counting at byte zero of the *file*. Look for **65 63 68 6F**, the ASCII bytes for **echo**. Look for **0D** after **online** and again after **pwd**. Each carriage return is one byte. The length shown as hex **17** in the directory agrees with 23 bytes, including those two line endings.

```
(Term!02)/H1/station:dir -e /h1/station

 Directory of /h1/station  2026/11/06 19:30

Owner  Last modified   Attributes Sector Bytecount Name
----- ---------------- ---------- ------ --------- ----
   0  2026/11/06 19:27  ------wr      28        17 report
   0  2026/11/06 19:27  ----r-wr      30         7 archive
   0  2026/11/06 19:27  ------wr      38        17 report.old


(Term!02)/H1/station:dump report

Address   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 2 4 6 8 A C E
--------- ---- ---- ---- ---- ---- ---- ---- ----  ---------------
00000000 6563 686F 2062 6561 636F 6E20 6F6E 6C69  echo beacon onli
00000010 6E65 0D70 7764 0D                        ne.pwd.

(Term!02)/H1/station:
```

Figure 4: The guest's extended directory and hex dump show a 17-hex-byte report whose last bytes are 70 77 64 0D, the word pwd and carriage return.

Walk from a device name to the first text byte

The table was decoded from the actual HD1 image used to make that screen. Numbers beginning with **$** are hexadecimal. The operating system assigns sectors as the disk changes, so use the chain of references when reading a different image. An extra file created earlier can move every later allocation.

Read this object	Find this field or entry	Follow it here on the example image
Volume description, logical sector 0	Three bytes at offset $08 name the root directory's descriptor	**00 00 18** → sector $18
Root directory descriptor at $18	Its first segment starts at $19 and spans seven sectors	Read root directory records beginning at $19
Root directory record named **station**	Last three bytes name its descriptor	**00 00 20** → sector $20
Station directory descriptor at $20	First segment starts at $21	Read 32-byte directory records beginning there
Station directory record named **report**	Last three bytes identify its descriptor	**00 00 28** → sector $28, also shown by the guest's detailed listing
Report descriptor at $28	File length is $17. First segment starts at $29 and spans seven sectors	Read only the first 23 bytes at $29
File data at $29	The bytes start with **65 63 68 6F** and end in **70 77 64 0D**	These are the same bytes printed by the guest's Dump

For a raw image with this filesystem starting at byte zero, sector $29 begins at byte **$2900**. The extent describes allocated sectors. The file-size field tells a reader where valid file content ends. Space reserved after a short file's last byte is not part of that file.

Inspect the sector on your CoCo

On a personal practice image, use the disk editor as a viewer. Enter **ded /h1@** at Shell+. The trailing **@** ** selects the entire device. The first screen identifies logical sector zero in the upper left. Use row and column labels to read the three bytes starting at $08. To follow their value, press** S**, enter the hexadecimal sector number at the sector prompt, and press Return. For example, our first jump was** 18**. Use** S** again to follow each link in the table, taking the values from your own disk. Directory records are 32 bytes each. If a name is farther down the directory, continue in its data segments.

At the sector containing the first bytes of **report**, compare the right-hand character display and the first 23 hex bytes with the file dump you saw earlier. The disk view can display 256 bytes even though this tiny file ends much earlier. Press **Q**, then **Y** at the quit confirmation to return to Shell+. These steps only read sectors. The editor's other modes can write raw disk structures. Use the navigation keys above for this exercise.

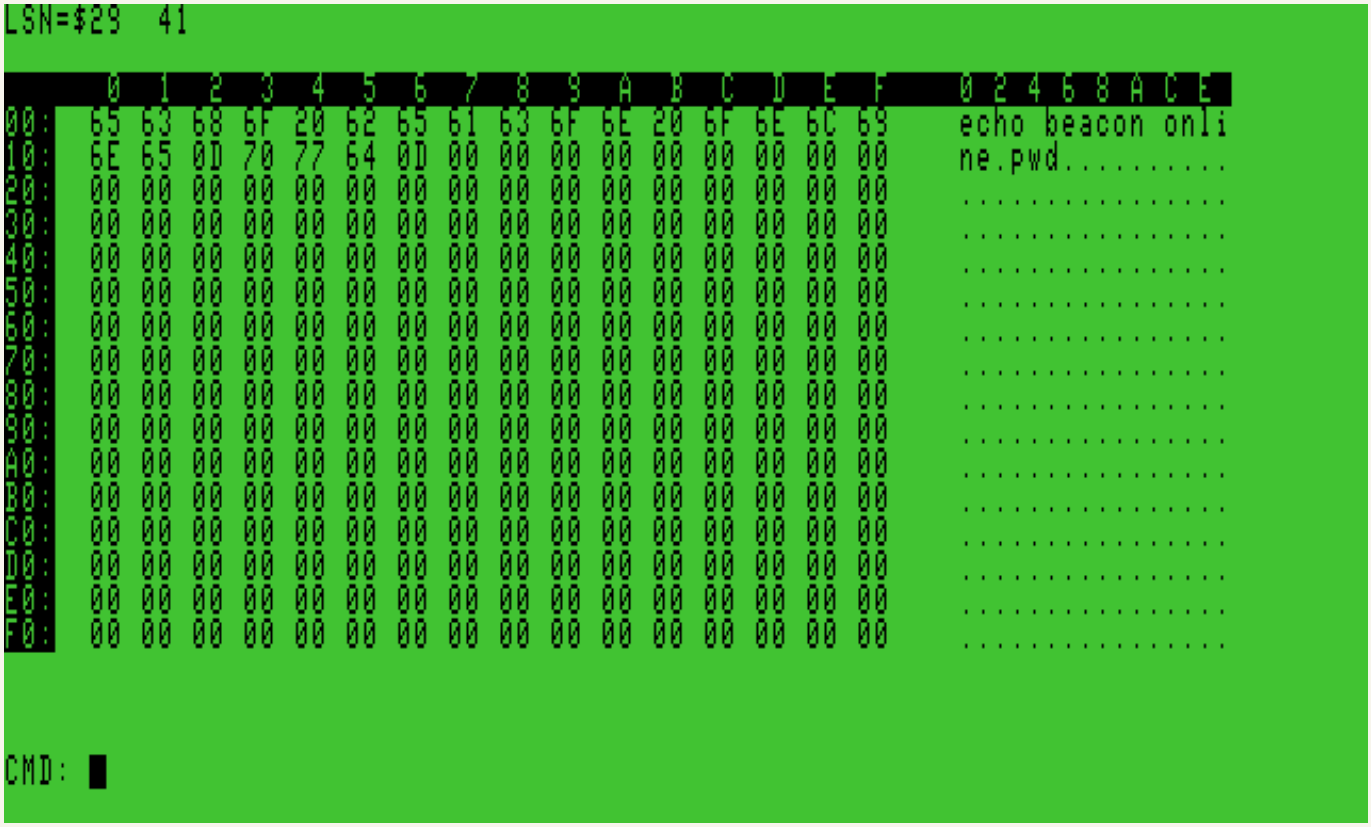


Figure 5: dEd shows sector $29, decimal 41. Its first two rows contain the station procedure and the rest of the displayed sector is zero in this copy.

Where the allocation map fits

Sector zero records the size of the map in bytes (**$1680**) and the number of sectors represented by one bit (**$0008**). $1680 bytes times eight bits per byte times eight sectors per bit accounts for the reported 368,640 sectors. Here the map starts at sector 1. A set bit marks an allocated cluster. The first bitmap byte in the pictured exercise was **FF**. It covered sectors $00 through $3F, including the volume structures and the practice files. When opening an existing file, RBF follows directory entries and segments. When it needs more space, the bitmap helps it locate free allocation units. A directory entry does not record a run of text sectors, and the bitmap does not say which filename owns a particular cluster. Together these structures answer different questions about the disk. The storage atlas gives the field offsets for the example.

Recover the right copy

Chapter 6 of 14

Suppose the relief operator remembers a report from the first shift, but the current disk also contains second-shift files. Recover the earlier state while keeping the later one. Give each state an unmistakable filename on the Mac and an observable difference inside OS-9.

Take a whole-image backup

Use your **personal practice** NitrOS-9 profile. At Shell+, finish the station commands from the previous chapters. Open **Media → Virtual Hard Disks…**, record the filename and location assigned to **HD1** and look for **Saves automatically**. The profile's **Files** list gives the path if the drive row only shows a filename. Wait for the save to finish. If you chose **Temporary changes**, use **Save As…** to retain those bytes and verify the actual HD1 assignment before continuing.

Quit the emulator. In Finder, duplicate the identified **HD1** image and name the duplicate **Station First Shift.vhd**. Keep it as the backup. The original remains at the pathname the practice profile knows. Copying a disk after the guest has stopped gives you a consistent point in time. The copy includes filesystem structures and the BASIC volumes that share this disk image.

Give the later state a recognizable marker

Reopen VCC Plus, load the same practice profile, run it and wait for Shell+. Type:

```
chd /h1/station
echo second shift >shift.two
list /h1/station/shift.two
```

The listing reads **second shift**. Let the automatic save finish and quit the emulator again. In Finder, rename the current, later HD1 image **Station Second Shift.vhd**. Keep it too. Duplicate **Station First Shift.vhd** and give that duplicate the *exact original HD1 filename* in the *original folder*. The profile can now open a writable restoration of your backup while both named snapshots remain intact. A different directory containing an identically named image will not satisfy the saved path.

Verify from a fresh boot

Open VCC Plus again, load the same practice profile, and choose **Run**. Verify the HD1 assignment. After OS-9 boots enter:

```
dir /h1/station
list /h1/station/shift.two
list /h1/station/report
```

The directory on the restored image contains **report**. Asking for **shift.two** gives **ERROR #216**, and the final listing shows **echo beacon online** and **pwd**. The missing marker is the desired result here. You just identified a generation of the disk by its contents. The later snapshot still holds the marker.

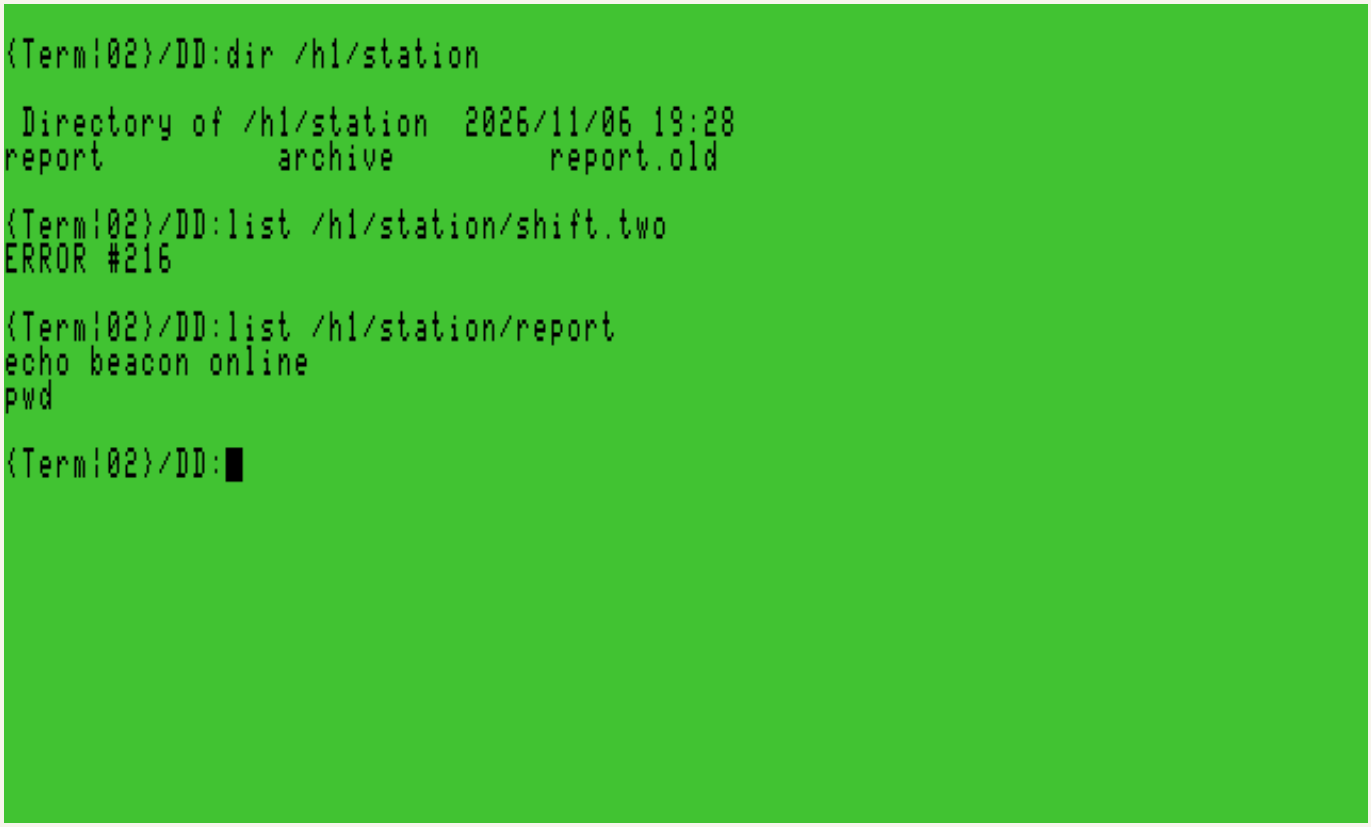


Figure 6: After booting the restored HD1, Shell+ lists the station directory, reports error 216 for the newer marker, and retrieves both original lines from report.

If the marker still exists, inspect the HD1 pathname and confirm which copy you duplicated. **If both paths are missing**, check whether you loaded fresh starter media or mounted another personal image. **If writes are pending**, finish or export them before replacing an image. A profile records the hardware and media assignments. The disk backup supplies the actual directories, permissions and file data for that point in time. This recovery leaves the shell working in **/DD** after boot. Use **chd /h1/station** before resuming exercises that name station files without their directory. You can keep the restored copy as the active practice disk for the remaining chapters.

Catch a process while it sleeps

Chapter 7 of 14

You can leave a job running on the CoCo and still type another command. Try this with a program whose only job is to wait. You will have time to find it in the process table, see who started it, and finish it deliberately.

What came back?

Begin at Shell+ in your practice session. Enter each line separately.

```
sleep 1
echo %*
list /h1/absent
echo %*
```

Use a practice disk where **absent** is an unused filename. After the short sleep, the first **echo** prints **000**. After **list** reports **ERROR #216**, the second prints **216**. **%*** expands to the last command's status in this Shell+. Inspect it immediately. Running another command can replace the evidence.

OS-9 gives each running instance of a program a **process ID**. When the shell starts an external program in the foreground, the child inherits paths such as standard output. The parent waits for the child to finish and can collect its exit status. A command that prints nothing can succeed, as Sleep just did.

Let two jobs overlap

```
sleep 3600 &
procs
```

The ampersand lets the parent shell accept another command while the child runs. The shell reports a number such as **&003**. Look in the **Primary Module** column for **Sleep**. Read across to **Id**, its own ID, and **PId**, its parent. During the illustrated run, Sleep was 3 and its parent was 2. Procs itself was another child of 2. It had to be running to report the table.

```
&005
&006
&007

Shell+ v2.2a 26/11/06 19:30:22

(Term!02)/DD:sleep 1

(Term!02)/DD:sleep 3600 &
&003

(Term!02)/DD:procs

          User                        Mem Stack
Id  PId Number  Pty Age Sts Signl Siz  Ptr   Primary Module
--- --- ------- --- --- --- ----- --- ----- ---------------
  2   1       0   128 128 $80     0   31 $63E2 Shell
  3   2       0   128 128 $C0     0   31 $60DF Sleep
  4   2       0   128 128 $80     0   31 $1EF3 Procs
  5   0       0   128 130 $80     0   31 $5CE2 Shell
  6   0       0   128 130 $80     0   31 $58E2 Shell
  7   0       0   128 129 $80     0   31 $53E2 Shell

(Term!02)/DD:
```

Figure 7: The guest process table shows Shell as ID 2, Sleep as ID 3 with parent 2, and the inspecting Procs command as ID 4 with the same parent.

The **Pty** and **Age** fields belong to the scheduler. A runnable process competes for processor time. A sleeping or waiting process can give up its turn until its event occurs. Giving two processes the same base priority does not make their progress identical. The hour-long sleeper spends its time waiting. The interactive shell can be waiting for a key at the same moment.

End only the child you started

Type **kill**, a space and the **Id you just observed for Sleep**. Press Return and run **procs** again. The Sleep row should disappear. Other Shell rows belong to the prepared system's other sessions. Leave those running.

The shell can report a child termination and error **228** as it collects the killed child. In this exercise that is the expected consequence of asking a process to stop. A signal and an exit status have different jobs. One delivers an event to a process that is still alive. The other tells a waiting parent the outcome after the child finishes.

At the operating-system interface, **F$Fork** creates a process, **F$Wait** lets the parent collect a child's termination, and **F$Exit** ends the calling process with its status in B. You will write that last call in the beacon report program. For the register contracts see The NitrOS-9 Project, *NitrOS-9 Technical Reference*, revision 0.1 (2004), pp. 99–100 and 128.

Where does a program live?

Chapter 8 of 14

A copy of Echo is on disk. There is also an Echo in the running system's module directory. Its instructions can serve one shell and then another. Each invocation still needs its own arguments, writable storage and stack.

Read a module's identity

```
ident /dd/cmds/echo
mdir
```

ident examines the named disk file. In the supplied edition, look for the name **Echo**, size **$0022** (34 bytes), execution offset **$0012** and a good module CRC. Offset $0012 means the first instruction is 18 bytes from the start of the module. The header tells the loader where to begin. The CRC checks the module's bytes against the check value recorded in the file.

mdir lists modules currently known to the kernel. The prepared boot loads a group of frequently used utilities, so Echo is already there. A file may contain several modules, each with its own header, name and entry in memory. A module can also be a file manager, driver or device descriptor. A name present in **mdir** alone tells you little about whether that name makes sense at a command prompt.

Count references you create yourself

Use a fresh practice boot with no other job using **Cmp**. Here Cmp is the small file-comparison program. Use the lowercase letters **mp** in the filter to match the ending of **Cmp** as it is printed.

```
load cmp
mdir -e ! grep mp
link cmp
mdir -e ! grep mp
unlink cmp
mdir -e ! grep mp
unlink cmp
mdir -e ! grep mp
```

Read the last number before the name. The four checks on our guest gave **1**, **2**, **1**, and finally no Cmp row.

```
Data Size:   $01C2    #450
Edition:     $05      #5
Ty/La At/Rv: $11 $81
Prog mod, 6809 obj, re-en, R/O

(Term!02)/DD:load cmp

(Term!02)/DD:mdir -e ! grep mp
:   9    0  27C 11  0 r   1 Cmp

(Term!02)/DD:link cmp

(Term!02)/DD:mdir -e ! grep mp
:   9    0  27C 11  0 r   2 Cmp

(Term!02)/DD:unlink cmp

(Term!02)/DD:mdir -e ! grep mp
:   9    0  27C 11  0 r   1 Cmp

(Term!02)/DD:unlink cmp

(Term!02)/DD:mdir -e ! grep mp
```

Figure 8: Four queries for Cmp show its use count climb from 1 to 2 and drop back to 1. The last query has no matching row.

load brought the program from the execution directory into memory and retained a reference. **link** found that named module in memory and retained a second reference. Each **unlink** released one. The disk file remains available after the module leaves memory. If other processes have acquired the module, their references also count. Release only the references you created. For boot modules, keep the system's existing references intact.

Give each caller somewhere to write

A re-entrant program can serve multiple callers using one copy of its code. Position-relative addressing lets a reference inside that code keep pointing to the same item after the module is placed in memory. Per-process data and a private stack give different executions a place to store different answers. Code that stores a changing counter directly into its shared instructions or embedded constants would break that arrangement.

The directory of running **processes** therefore answers a different question from the directory of **modules**. In the preceding chapter you found several Shell processes. In the next chapter you will see several of them map the same physical code block. For the header layout and module CRC see The NitrOS-9 Project, *NitrOS-9 Technical Reference*, revision 0.1 (2004), chapter 3, pp. 26–32.

One address, two views

Chapter 9 of 14

Two Shell processes can have different private data while executing the very same instruction bytes in RAM. The CoCo 3 makes that possible with its memory management unit. NitrOS-9 Level Two keeps the appropriate block assignments for each process and installs the mapping it needs when switching execution.

Read a process map

```
pmap
mfree
```

Read **pmap** across. Each pair in the heading names the leading hexadecimal digits of one 8 KiB logical range. **01** covers $0000–$1FFF. **23** covers $2000–$3FFF. The last column, **EF**, covers $E000–$FFFF. The numbers in each row are physical block numbers. Dots mark positions without a block assigned to that process by OS-9. In the example, Shell processes 2 and 5 have different blocks at the left, **07** and **0D**, and the same block **0A** at the right. They can each have a private stack or data while sharing their code. **mfree** instead counts available physical memory in system-wide blocks. A 512 KiB CoCo has 64 blocks of 8 KiB. Installing another block somewhere in that machine does not give one 16-bit process a ninth slot in its logical map.

Predict an instruction’s physical address

Use the supplied system and the two-line station procedure you made earlier. In the station directory, inspect Shell's placement and its entry offset.

```
mdir -e ! grep hell
ident /dd/cmds/shell ! grep 0074
```



Figure 9: PMap shows multiple Shell processes with top block 0A. The module directory places Shell at offset zero in that block, and Ident reports an entry offset of $0074.

Here Shell starts at offset zero in physical block **$0A**. The **Shell** header in the supplied file has execution offset **$0074**. Since that block occupies the top slot in the process, its entry is **$E000 + $0074 = $E074**. If the image or Shell edition differs, inspect the header and map for that edition before setting a breakpoint. A file with multiple modules can produce several sets of header information.

Quantity in this run	Value
Start of physical block $0A	$0A × $2000 = $14000
Offset into the physical block	$0074
Predicted physical instruction address	**$14074**

Stop while that map is active

1. Open the machine **Debugger**. In **Breakpoints**, add the logical address **E074**. Resume if you paused to enter it.
2. At the CoCo shell, run **shell report** in the station directory. The child shell reaches its entry instruction and the emulator stops there.
3. Read **PC** and the **MMU** inspector. Our run stopped at **E074** with **task 1** active. The effective bank for logical $E000 was **0A**. A stop inside an interrupt handler or a different process can show a different map.
4. In **Memory**, inspect **Mapped CPU** at **E074**, then **Physical RAM** at **14074**. Both views began **32 7B 34 36 30 C9 18 15**. Checking eight bytes at a predicted address gives you a better fingerprint than one matching byte.
5. Remove the breakpoint and **Continue**. The procedure prints **beacon online** and its working directory, and the parent prompt returns.

A debugger breakpoint is specified in logical CPU space. Context matters if another process also executes at the same address. The GIME's two hardware task selections do not limit the OS to two processes. OS-9 can reload registers from saved mappings as processes take turns. When a PMap position is unused, a hardware register may still show a value programmed by the OS. Use the process map to interpret ownership and the debugger's effective mapping to interpret a particular CPU read.

Book 10 shows the same translation with a byte deliberately written by a BASIC-hosted program. For the CoCo 3 MMU under Level Two see The NitrOS-9 Project, *NitrOS-9 Technical Reference*, revision 0.1 (2004), chapter 2, p. 13.

Open a device by name

Chapter 10 of 14

Imagine removing the output cable from a terminal and plugging it into a disk. Programs would need radically different electrical signals for those two devices. OS-9 gives a program a simpler operation. Write bytes to an open path. The system finds the right file manager and driver for that path. See the dispatch in the live device table.

Catch the drive while it is open

Use a fresh filename on HD1. From Shell+ enter:

```
devs >/h1/devices
list /h1/devices
dmode /h1
```

There is a reason to save this particular report. While **devs** examines the active device table, its output file holds H1 open. If you just run **devs** at a prompt, an idle H1 may be absent from the display. The saved report records the conditions at the instant the utility ran.

```
(Term!02)/DD:mdir -e ! grep mp
:    9     0  27C 11   0 r    1 Cmp

(Term!02)/DD:unlink cmp

(Term!02)/DD:mdir -e ! grep mp

(Term!02)/DD:devs >/h1/devices

(Term!02)/DD:list /h1/devices

Device Table at: 6B00

Dv Desc  Drvr    Stat  File  Usr
Nm Port  Name     Mem  Mgr   Cnt
-------  -------  ----  ---
DD FFE0  EmuDsk 6900  RBF    06
Te FFA0  VTIO   6800  SCF    04
W2 FFA2  VTIO   5A00  SCF    03
W3 FFA3  VTIO   5900  SCF    03
W4 FFA4  VTIO   5000  SCF    02
H1 FFE0  EmuDsk 6900  RBF    01

(Term!02)/DD:
```

Figure 10: The guest's saved device table pairs DD and H1 with EmuDsk and RBF. Term and the window devices use VTIO and SCF.

Look for **H1**, **EmuDsk** and **RBF** on one row. The descriptor names the device and specifies its driver, file manager and operating parameters. **RBF**, the random block file manager, understands directories, files and disk sectors. **EmuDsk** is the driver that talks to the emulated virtual disk controller. In this profile, **dmode /h1** reports drive number **01**, and VCC Plus assigns the second working image to HD1. A different controller can use a different driver under an RBF filesystem.

Trace three paths from the same shell

If the program sends bytes to…	Follow the request
A file under **/h1**	IOMan dispatches through RBF. RBF resolves the file and its sectors. EmuDsk transfers blocks through the virtual hard-disk device to the image assigned to HD1.
The terminal **/Term**	SCF handles a stream of characters and its terminal options. The VTIO driver and the CoCo keyboard and display components deliver the interaction. Read the Term row next to H1 in the figure.
A shell pipeline	PipeMan manages a buffered stream with the supplied Piper driver. The writer sends bytes to one end. The reader takes them from the other. Closing the last writer lets the reader detect the end of the stream.

The program refers to an open **path number** during I/O. It need not resolve the pathname or understand sectors each time it writes a line. The shell can arrange path 1 before launching the child, which is how the same program can write its report to any of those three destinations. A **device descriptor** is a named module that tells OS-9 how to reach a device. An **open path** holds the state of a particular opening, such as a file's current position and access mode.

The graphic panels in VCC Plus show which host images are attached to the emulated drives. The guest device table shows the software OS-9 is using to reach them. Compare the quick-start equipment map with the same two assignments in your current profile. For the division of responsibilities see The NitrOS-9 Project, *NitrOS-9 Technical Reference*, revision 0.1 (2004), chapters 4–7, pp. 33–86.

Who owns this terminal?

Chapter 11 of 14

OS-9 can run several shells on one CoCo 3. In the prepared boot, Shell+ starts on **/Term** and additional shells run on window devices. All of them share one physical keyboard. OS-9 chooses which guest window receives it. macOS separately decides whether your typing reaches the emulator at all.

Identify your terminal

Click the CoCo display and enter **tmode**. The first line on the main console is **/Term**. The options below include **eko** for character echo, **eor** for the end-of-record character, and **eof** for the end-of-file character. Here **eor=0D** is a carriage return. These are options on the current terminal path. You can inspect them without changing them. Run **devs** to inspect the active window devices. On the supplied profile, the startup procedure uses W2, W3 and W4. Use the practice system with **W7** free for this exercise. If W7 is already in use by your work, choose a fresh practice session. A window may display output even when it has no shell reading the keyboard.

Draw the station’s signal meter

At the main terminal enter the following two commands. **iniz** holds an attachment to the new device. **wcreate** defines its screen and colors. Screen type **8** selects a 320 × 192, 16-color graphics screen. The rectangle in the creation command is measured in character cells. It starts at the top left and spans 40 columns by 24 rows. The final three values select the foreground, background and border palette entries.

```
iniz /w7
wcreate /w7 -s=8 0 0 40 24 03 00 00
```

Send three pairs of drawing commands to W7 while you are still typing in /Term. The numbers after **display** are hexadecimal bytes. **1B 40** sets the draw pointer, and **1B 4A** fills a rectangle from that pointer to the supplied opposite corner. Each coordinate consists of a high byte and a low byte. The driver interprets positions under its current scaling. The three bars share the same lower edge at Y = $0088.

```
display 1b 40 00 60 00 68 1b 4a 00 a0 00 88 >/w7
display 1b 40 00 c0 00 48 1b 4a 01 00 00 88 >/w7
display 1b 40 01 20 00 28 1b 4a 01 60 00 88 >/w7
display 1b 21 >/w7
```

The last command selects W7 for display. Look for three bars rising from left to right. In the captured palette the bars are green and the background is white. The staircase is the useful test even if your colors differ.



Figure 11: The actual guest graphics window displays three solid green bars. Their lower edges align and each successive bar is taller.

Return to the shell and release the window

With the guest display focused, press **CoCo Clear**. VCC Plus maps the Mac **Home** key to Clear. On the prepared profile we returned to the original terminal in one press. If your active windows are ordered differently, cycle until you see the original **Term** prompt. Trying to type a cleanup command on the graphic would send keys to the wrong window.

Back on /Term enter:

```
display 1b 24 >/w7
deiniz /w7
tmode
```

1B 24 ends the display window. **deiniz** releases the attachment you took with **iniz**. **tmode** should again identify **/Term**. You have created a graphics device, drawn into it, selected it, regained keyboard control and released the device while the same guest operating system continued running.

A second OS-9 window still belongs to the CoCo. The optional guest connection to the Mac shell crosses into macOS and has its own permission control. Use the terminal name and the command interpreter to decide which machine will receive your next command.

For the window protocol, see The NitrOS-9 Project, *The NitrOS-9 Level 2 Windowing System*, pp. 7–8 (creation), 20–23 (end and setup), 44 (selection), 49 and 58 (bar and draw pointer).

Put the night watch on duty

Chapter 12 of 14

Your station needs someone to count three watches. In BASIC09 you can give the job a name, declare the storage it needs, and call it from either the language workspace or the shell. The same three lines of output will tell you when each version works.

Write the procedure on HD1

Start at Shell+ with the writable practice disk in HD1. Use new names for **night.b09** and **night** so you can keep both the original and the packed copy. Enter one command per Return.

```
chd /h1
build night.b09
```

BUILD prompts with a question mark for each new line. Enter these six lines, then press Return at the empty prompt. The file is also available as night.b09.

```
procedure night
dim watch:integer
for watch=1 to 3
print "watch ";watch
next watch
end
```

The first line gives the procedure the name **night**. Its integer variable **watch** belongs to this procedure. The **for** assigns 1 before the first trip through the loop. Each **next watch** advances the counter and tests whether another iteration is due. Each trip prints a word and the current value on one line. **end** finishes the procedure. The loop has a name and a matching end point on the page, so you can follow it without line numbers.

Find it in the BASIC09 workspace

At Shell+ make the practice disk your execution directory and launch the language system from the system disk.

```
chx /h1
/dd/cmds/basic09
```

There are two independent locations here. The data directory is where BASIC09 opens **night.b09**. The execution directory determines where the packed version will be written. The absolute name **/dd/cmds/basic09** still locates the language system after you change directories.

Look for **Ready** and the **B:** prompt. Enter:

```
load night.b09
run night
```

LOAD reads the text file and recognizes the procedure by the name inside it. RUN takes that procedure name. You should see:

```
watch 1
watch 2
watch 3
```

Ready and **B:** appear again. You are still in BASIC09. Compare the way you entered the source with programming in CoCo BASIC. On this disk BASIC09 is an OS-9 program with its own workspace and commands. The CoCo BASIC editor in the Mac Develop menu works with the ROM BASIC interpreter.

Hand it to the runtime

At **B:** enter the next two commands.

```
pack night
bye
```

PACK writes a module containing the procedure's intermediate code. BYE leaves the language workspace. Now launch the smaller BASIC09 runtime from Shell+.

```
/dd/cmds/runb night
```

This time you should see three watches and then the **OS-9 shell prompt**. RUNB looked up the packed procedure by name and ran it without opening the editing workspace. In this configuration giving RUNB the full path as its procedure argument produced error 043. Use the execution directory and the procedure name shown above.



Figure 12: The green guest display shows three watch numbers from BASIC09, the pack and bye commands, and the same three numbers from RUNB at the shell.

Restore normal command lookup when you finish.

```
chx /dd/cmds
dir /h1
```

Find **night.b09** and **night**. Keep the first file to edit the original text. The second can run under RUNB. In the next chapter you will give the station an executable module whose body consists of 6809 instructions.

For the language and its workspace, see Microware Systems Corporation, *BASIC09 Programming Language Reference Manual*, revision H (1984), “Introduction to BASIC09 Programming” and “Program Statements and Structure.”

Assemble a program on the CoCo

Chapter 13 of 14

Write a station report in 56 bytes. Most of those bytes will describe the program or hold its message. Only seven instructions are needed to deliver the line and return control to the shell.

Give OS-9 a module

Begin at Shell+ with **/dd/cmds** as the execution directory and a personal HD1. Keep any previous **beacon.a** or **beacon** by using a fresh practice image for a repeat. At the shell enter:

```
build /h1/beacon.a
```

Type the Beacon source one line at a time. Start each label, such as **start** or **finish**, in the first column. Keep a leading space on lines that start with an assembler directive or an instruction. Finish by pressing Return at the next empty BUILD prompt.

```
        mod endmod,name,$11,$81,start,256
name    fcs /beacon/
        fcb 1
start   lda #1
        leax note,pcr
        ldy #notesiz
        os9 $8c
        bcs finish
        clrb
finish  os9 $06
note    fcc /beacon report/
        fcb 13
notesiz equ *-note
        emod
endmod  equ *
        end
```

mod starts a module. Its arguments supply the end of the module, the name, its type and language, its attributes and revision, the entry point, and a request for working storage. The assembler resolves the labels as offsets. **fcs** stores the module name with the high bit of its last character set. The byte immediately after the name gives edition 1. Those bytes belong to the name and header information. The first executable instruction begins at **start**.

The header declares a 6809 program module ($11), re-entrant with revision 1 ($81). OS-9 can share its program bytes among callers. Each process gets its own working storage and stack. Beacon requests 256 bytes and leaves its message inside the read-only module. **fcb 13** adds the carriage return expected by the line-writing service. The assembler evaluates **notesiz** as 14, the length including that final byte.

Follow the two system calls

At this line	What the register or flag means
lda #1	A selects path 1, the process's standard output. The shell chose what is on the other end.
leax note,pcr	X becomes the address of the first message byte. The displacement is relative to the program counter, so the module can execute at different logical addresses.

At this line	What the register or flag means
ldy #notesiz	Y supplies the maximum number of bytes the write may consume.
os9 $8c	I$WritLn asks the operating system to write a line. The assembler emits a 6809 SWI2 instruction followed by the service number.
bcs finish	The service sets carry on failure and puts the error number in B. This branch preserves the number for the caller.
clrb	On the success path the exit status becomes zero.
os9 $06	F$Exit ends the process and passes the status in B to its parent.

The CPU never has to know an address for the screen. In the BASIC ROM example, a caller used a ROM vector to send a character. Here the OS-9 process selects a path and asks the kernel for a service. In the next exercise the shell will put a disk file on that very same path.

Let the guest assembler check your work

```
list /h1/beacon.a
asm /h1/beacon.a o=/h1/beacon
ident /h1/beacon
```

Use the listing to catch a missing label or misplaced space before assembling. In the **asm** command **o=** names the output file. The final **emod** in the source generates the closing CRC. Here are four checks on what came out:

Field from IDENT	Expected value	How to read it
Header name and module size	**beacon**, **$0038**	The whole module, including header and CRC, occupies 56 bytes.
Execution offset	**$0014**	The first instruction is 20 bytes from the beginning of the module.
Data size	**$0100**	The process requests 256 bytes for working storage including its stack. These bytes are allocated at execution.
Module CRC	**$3D8059 (Good)**	IDENT recomputed the CRC successfully. This particular value identifies the bytes produced by the printed listing.

```
        emod
endmod  equ *
        end

(Term!02)/H1:asm /h1/beacon.a o=/h1/beacon
00000 error(s)
00000 warning(s)
$0038 00056 program bytes generated
$0000 00000 data bytes allocated
$00C2 00194 bytes used for symbols

(Term!02)/H1:ident /h1/beacon

Header for:  beacon
Module size: $0038    #56
Module CRC:  $3D8059 (Good)
Hdr parity:  $10
Exec. off:   $0014    #20
Data Size:   $0100    #256
Edition:     $01      #1
Ty/La At/Rv: $11 $81
Prog mod, 6809 obj, re-en, R/O

(Term!02)/H1:
```

Figure 13: The guest assembler reports zero errors and zero warnings. IDENT shows the 56-byte beacon module, entry offset $0014 and good CRC $3D8059.

The assembler's line counting allocated data can read zero even though the header requests $0100 bytes at runtime. Compare what was **written into the file** with what a **new process will need**. Now take your completed module to the station console.

For the format and service conventions, see The NitrOS-9 Project, *NitrOS-9 Technical Reference*, revision 0.1 (2004), chapter 3, pp. 26–32, F$Exit, p. 99, and I$WritLn, p. 148.

Deliver the beacon report

Chapter 14 of 14

You have a file that contains a program. The filesystem controls who may execute the file. Inside the file, the module header tells OS-9 what it can load. Those are two different checks. Give the file execute permission and run it from its full path.

Choose the destination

```
attr /h1/beacon e
/h1/beacon
```

The program prints **beacon report** and the shell prompt returns. Now run it again. This time let the shell create a fresh file named **report** at the root of your practice HD1.

```
/h1/beacon >/h1/report
list /h1/report
```

The first command goes straight back to the prompt. The second displays the two words. The file **/h1/report** contains the message from the module. Your shell procedure from the beginning of the book lives one directory deeper under **/h1/station/report**.

Trace one line through the system

Stage	On the terminal run	On the file run
Shell+ prepares the child	Standard output refers to the terminal.	The shell opens the named output file and supplies it as standard output.
The module begins	A new process starts at the entry offset in Beacon's header.	The same module, entry point and instructions are used.
Beacon calls I$WritLn	A holds 1, X points to the text and Y holds 14.	A, X's meaning and the byte count are the same.
OS-9 resolves path 1	The path leads through the character file manager and terminal driver.	The path leads through RBF and the HD1 disk driver.
Beacon finishes	F$Exit returns the status to its parent shell.	The process exits and the saved file can be opened independently.

The number **1** in the program identifies a slot in *that process's* path table. It does not permanently mean screen or disk. That small indirection explains why a program written by someone who never saw your practice disk can still write a file there.

Catch the error at the right layer

With **/h1/report** still on the practice disk, enter exactly the same redirection and then read the existing file.

```
/h1/beacon >/h1/report
list /h1/report
```

The repeated create reports **ERROR #218**. The listing still prints **beacon report**. The shell needed to open the destination before launching the child. When that open failed, no Beacon instruction ran. A program's carry check after I$WritLn handles a different case, a failure *after* a process has started and requested a write.

Give the line to another program

Can you display only matching lines without changing any source or creating another report on disk? Predict what **grep** will receive, then try:

```
/h1/beacon ! grep report
```

The guest's GREP displayed **: beacon report**. The leading punctuation belongs to the filter's display of the match. The module supplied the same line it wrote to disk. In this run the shell connected Beacon's output to a pipe and the filter read the other end. The filter still had its own output path to the terminal.

```
Ty/La At/Rv: $11 $81
Prog mod, 6809 obj, re-en, R/O

(Term!02)/H1:attr /h1/beacon e
--e-rewr

(Term!02)/H1:/h1/beacon
beacon report

(Term!02)/H1:/h1/beacon >/h1/report

(Term!02)/H1:list /h1/report
beacon report

(Term!02)/H1:/h1/beacon >/h1/report
ERROR #218

(Term!02)/H1:list /h1/report
beacon report

(Term!02)/H1:/h1/beacon ! grep report
: beacon report

(Term!02)/H1:
```

Figure 14: The guest runs Beacon on the terminal, routes it into a file, catches error 218 on a repeated create, reads the intact file and finds the report through a pipeline.

You can now explain a trip that begins at a key and ends in a disk sector. Shell+ interprets the command, opens a path and starts a process. The process requests a service. A file manager interprets that request for this kind of device, and its driver transfers the bytes. Back at Shell+, **list** is a different process opening that same file for reading. For another way to make an OS-9 program take part in this arrangement, try the period C compiler.

Find a command

Appendix A

You remember the job, or you remember the name. Start with either. The spellings below describe the prepared NitrOS-9 Level Two image in VCC Plus. A different boot disk can supply a different shell, command set or edition of a tool.

Start with the job

I want to…	Start here
Find a file or figure out which directory the shell is using	Working and execution directories, then `dir`, `pwd`, `pxd`, `chd`, `chx`
Save several commands and run them together	Shell procedures, then `build`, `list`, `shell`
Send the same text to the screen and a file	Paths and pipes, then `tee`
Compare two versions before replacing a file	Recover a copy, then `copy`, `cmp`, `rename`
See who's using the CPU or holding memory	Processes, modules and address maps
Identify a disk or change a terminal option	Devices, terminals, then `devs`, `dmode`, `tmode`
Run my own program inside OS-9	BASIC09, assembly and the language guide

How to read a command form

Words in italics stand for your own names. Square brackets enclose optional operands. A *device* is a guest name such as /h1. A *path* can also name a file or directory on that device. A *module* is identified by its internal name. For example, `ident /h1/beacon` reads a file, while `link beacon` asks for a module by name. The module chapter explains when that name becomes available.

Shell+ interprets operators before it starts the command. A single > creates a new output file, `!` connects two programs, and & starts a background job. A procedure is text read by a shell. A command can also be built into the shell or supplied by an executable module, including one loaded at boot. Looking only in /DD/CMDS will miss some names the shell accepts.

The compact forms below cover the guest help catalog. A lookup entry identifies an operation. The linked lessons give inputs and visible results for the operations used in the book. For edition-specific switches, use the guest's HELP entry for the named command. Names such as GFX and INKEY belong inside a BASIC09 procedure.

Alphabetical finder A–D

Name and form	What it does
ASM *source* [o=*module*]	Translate assembler source. Read the reported error count before attempting to execute the output.
ATTR *path* [*permissions*]	Inspect or change permission bits. Adding `e` gives the owner permission to execute a file.
BACKUP [*source-device destination-device*]	Duplicate a volume at the device level. The destination contents are replaced. See the working-image recovery exercise for a Mac-side alternative.

Name and form	What it does
BASIC09	Open the programming workspace at its B: prompt.
BINEX *input output*	Convert binary data to Motorola S-record text for transfer. EXBIN performs the reverse conversion.
BUILD *path*	Make a short text file from lines you enter. A blank Return ends entry.
CHD *directory* · CHX *directory*	Change the current data directory or the command execution directory respectively.
CMP *first second*	Compare files byte by byte and report differences.
COBBLER *device* · CONFIG	Construct boot media. Work on a spare image with an independently bootable original.
COPY *source destination*	Duplicate a file under a fresh name.
CPUTYPE · DATE	Report the guest processor or the guest date.
DCHECK *device*	Check the relationships among a disk's allocation map, directories and files.
DEBUG · DED *device*	Inspect machine execution or edit raw disk sectors. A raw write can change several directory entries at once.
DEINIZ *device*	Release a device attachment. INIZ acquires one.
DEL *path* · DELDIR *directory*	Remove a file, or remove a directory tree and its contents.
DEVS	Show the active devices and their drivers and file managers.
DIR [-e] [*path*]	List a directory. The extended listing shows ownership, attributes and sizes.
DIRSORT [*directory*]	Reorder entries in the directory itself. Use a copy if the original order matters.
DISASM [*path*] · DISPLAY *codes*	Decode instructions, or emit character values such as terminal controls.
DMEM [*block*] *offset* [*length*]	Inspect bytes from a guest memory block.
DMODE *device* [*settings*]	Read or change an RBF disk descriptor's settings.
DSAVE [*destination*]	Generate a procedure that copies a directory hierarchy. Inspect the generated commands before running them.
DUMP *file*	Show the offset, hexadecimal bytes and printable characters side by side.

Alphabetical finder E–M

Name and form	What it does
ECHO *words*	Write a line on standard output. The shell can redirect it.
EDIT [*file*] · MINTED [*file*]	Edit text using the editor supplied by the guest. BUILD suffices for all required source entry here.
ERROR *number*	Look up a guest error number. Read it alongside the command that produced it.
EX *module*	Replace the current shell process with the named program.

Name and form	What it does
EXBIN *input output*	Decode an S-record file into binary.
FORMAT *device*	Initialize a filesystem. An existing directory and its data can be lost.
FREE *device*	Report the volume's geometry and available storage.
GFX · GFX2 · GRFDRV	The first two are BASIC09 graphics interfaces. GRFDRV supplies lower-level window drawing services.
GREP *pattern*	Select matching input lines. The pipe exercise supplies its input.
HELP [*topic*]	Consult the help text installed on the guest disk.
IDENT *module-file*	Read module identity, revision, length and CRC information from a file.
INIZ *device*	Attach a device so that the system can keep it initialized.
INKEY	Read a keystroke through a BASIC09 subroutine.
IRQS	Inspect the guest interrupt service table.
KILL *process-ID*	Send an abort signal to the selected process. Use the ID from the current PROCS display.
LINK *module* · UNLINK *module*	Acquire or release a reference to a module already known to the kernel.
LIST *path*	Send a text file's contents to standard output.
LOAD *path*	Bring one or more modules from a file into memory.
LOGIN	Start a login dialogue using the guest account configuration.
MAKDIR *path*	Create a directory entry and its directory file.
MDIR	Enumerate the modules the kernel currently knows.
MEGAREAD	Measure sustained reads from a supplied source.
MERGE *file …*	Send the files' contents in sequence to standard output.
MFREE · MMAP	Report available physical memory or map the system's memory blocks.
MODPATCH *patch-file*	Apply a prepared set of module changes.
MONTYPE · MPI	Inspect or select monitor type, or inquire about a Multi-Pak. These guest utilities have their own hardware assumptions.

Alphabetical finder O–X

Name and form	What it does
OS9GEN *device*	Construct a disk bootstrap. Match the boot track and modules to the target hardware.
PADROM *size file* · PARK *device*	Prepare a padded ROM file, or park heads on a compatible physical drive. VCC Plus VHDs have no mechanical heads.
PMAP · PROC · PROCS	Inspect a process's memory map or the system's process list.

Name and form	What it does
PROMPT *message*	In the supplied help, the PROMPT utility displays a message and waits for a key.
PWD · PXD	Print the full path of the shell's data or execution directory.
REBOOT	Restart the guest. Allow writable media to finish saving first.
RENAME *old-path new-name*	Change the final name of a directory entry.
RUNB *procedure-name*	Run a packed BASIC09 procedure. See the Night procedure.
SAVE *file module …*	Write named resident modules to a file.
SETIME [*date/time*] · SETPR *ID priority*	Set guest time or change the scheduling priority of a process.
SHELL [*procedure*]	Start an interactive child shell or have it interpret a procedure file.
SLEEP *ticks*	Suspend the caller for a specified interval. Used as an identifiable background child in this book.
SMAP	Display a system memory map.
TEE *path*	Copy standard input to standard output and another destination.
TELNET *host port*	Connect a guest terminal session. See the Mac permission dialogue.
TMODE [*options*]	Inspect or change an open terminal path's options.
TOUCH *file*	Update the date associated with a file.
TSMON [*terminal*] · TUNEPORT *device*	Manage waiting terminal sessions or adjust a supported serial port.
VERIFY [*options*]	Check module integrity. Inspect update options before applying them to a file.
WCREATE [*options*] · XMODE *device* [*settings*]	Set up guest windows or inspect/change the defaults in an SCF device descriptor.

Explore the rest of /DD/CMDS

The supplied image contains 398 named directory entries here. A filename alone cannot tell you whether the entry is a shell command, an application, a data file or a bundle of modules. For instance, `DIR`, `DIR.NEW` and `DIR.ORIG` are three different entries. The system help catalog includes names that live elsewhere or are built into a shell. Use the finder above for that catalog. Use the directory below when an unfamiliar name turns up on this particular disk.

The following groups locate additional entries by task. Presence in this inventory means the name was read from the prepared filesystem. An executable module still needs compatible hardware, any companion files and the correct invocation. For an unfamiliar file, use the extended listing to check its type and permissions, and IDENT to inspect a module header. Consult the guest's /SYS/MAN and /SYS/helpmsg for available instructions.

Look under	Additional names on this image
File handling, conversion and disk tools	`ar`, `bfre`, `bfree`, `bigfile`, `cat`, `copy2`, `cp`, `d`, `ddmaker`, `dir.new`, `dir.orig`, `dloadx`, `dloady`, `dname`, `exttran`, `filepick`, `filer`, `files`, `fixtext`, `fmap`, `gcpy`, `ln`, `lof`, `ls`, `mscopy`, `mv`, `pcdos`, `pcsave`, `readdisto`, `rescue`, `rsdos`, `rssave`, `sdc`, `sortdir`, `tar`, `transfer`, `trscopy`, `unzip`, `vdisk_mods`

Look under	Additional names on this image
Text, tabulation, editing and formatting	adj, cut, cuts, dict, diff, fgrep, gsort, lninput, lower, macro, maketxt, more, mroff, nroff, qsort, scred, sled, sort, spell, split, strings, tail, text, tsed, tsedit, txt80x25, ul, uniq, unwords, upper, wc, words
Assembling, compiling, linking and debugging	basecon, c, c.opt, c.pass1, c.pass2, c.prep, cc, cmdgen, crcoff, crcon, ipatch, make, makpatch, modbuster, rlink, rma, sleuth, syscall
Pascal and other language systems	kcbasic, pacos9, Pascal, Pascal_Compiler, PascalE, PascalErrs, PascalN, PascalS, PascalT.MODL, PascalT.PRUN, PCODEF
System, boot, accounts and configuration	adduser, alias, at, bios, bios3, chown, config.desk, config.desk.save, control, cron, date.orig, datetime, drivewire_mods, dtime, expire, fix, fstat, getclock, ide_ff50_mods, ide_ff60_mods, initsub, l2initsub, logname, mkdir, mkopt, multistart, os, os9gen.orig, ostart, printer_mods, profile, reset, setime.orig, setype, shell_21, sysdev, turnkey, user, utilpak1
Windows, graphics, fonts and printing	artist, bpix, cls, cm32vef, cocopr, color, desk, desk.old, desk.patch.scr, fixpix, g320c16, g640c2, g640c4, gclock, gif2vef, gport, gprint, gshell, guib, lpic, lpr, make80, makegshell, makegw, paint, pix, play, rleconv, rleview, scrn, vef2gif, vefio, view, viewgif, window, window.glr4, window.t32, window.t80b, window.t80s, wpix
Communications, mail and file transfer	accdata, area, areacode, autodial, crypt, httpd, inetd, kermit, mail, osterm, pop, postnews, readnews, remctrl, rmail, rnews, supercomm, telecom, uploadx, uucico, uucp, uudecode, uuencode, uuxqt, xmodemup, xymodem
DeskMate, calendar and office software	calendar, dm3, dmalt, dmalt.hlp, dmapps, dmbackup, dmcaln, dmcaln.hlp, dmcards, dmcards.hlp, dmdskini, dmhelp, dmledger, dmledger.hlp, dmload, dmmenu, dmmenu.hlp, dmpaint, dmterm, dmterm.hlp, dmtext, dmtext.hlp, dynacalc, gcal, gcalc, jdate, ledger, month, zone
Games, music and demonstrations	ball, banner, Bounce, cave, caveman, cookie, cursive, demo, des1, des2, des3, des4, des5, des6, dungeon, flight, Fran, ICONS, klondike, koronas, lander, maze, meep, MIDI.dddr, play.orig, playmus, quest, questdvr, questm, rogue, SerMIDI, sound, spals, speech, tank
Other historical applications, scripts and companion files	autoex.count, automode, dmpaint.hlp, dmtim, dw, ed, ezgen, gost, hdlns, hisub, lha, mgt, movsub, pag, pak, pmpts, PROCTAB, qtip, rep, rsb, savsub, sdcdriver, show, solution, Support, Support1, Support2, swread, tree, tspars, UBox3, Um3B, Um3D, Um3E, Um3K, Um3L, Um3M, Um3P, Um3S, Um3T, Um3U, Um3W, UMusE3, vu, xcom9, xlh

For a new system disk, inspect its own command directory and module list. Historical program collections often package multiple editions together. The tested command in a lesson is always identified by the guest and its visible result.

Device and storage atlas

Appendix B

Use this atlas when a name, a sector number and a Mac file all seem to point to the same thing. Each locates an object at a different layer.

Trace a request to its destination

Guest name or handle	OS-9 route on the supplied system	Destination
/dd	Default system disk descriptor through RBF	The boot filesystem supplied on HD0
/h0	H0 through RBF and EmuDsk, drive 0	The HD0 image shown under Virtual Hard Disks
/h1	H1 through RBF and EmuDsk, drive 1	Your writable HD1 image, with an initialized OS-9 filesystem
/Term	Term through SCF and VTIO	The guest terminal displayed in the emulator
A shell pipeline **!**	Pipe manager, with an open write end and read end	Bytes passed from one guest process to the next
Standard output, **1**	Per-process path number, inherited or redirected by the shell	Terminal, file or pipe selected for this run

devs reports relationships among active devices, drivers and managers. **dmode /h1** identifies the disk descriptor and its drive number. **tmode** inspects the current terminal options. The quick-start equipment map pairs the drive window with guest names. The HD1 virtual device serves sectors. It is RBF that interprets them as directory entries and file descriptors.

Decode the station disk

All offsets below are hexadecimal byte offsets *inside the named structure*. Multi-byte integers here are most-significant byte first. The example values come from chapter 5's actual exercise disk.

Structure	Offset and width	Value in the example
Volume description	$00, three bytes, total sectors	05 A0 00 → 368,640 sectors
Volume description	$04, two bytes, allocation map length	16 80 → 5,760 bytes
Volume description	$06, two bytes, sectors per cluster	00 08 → 8
Volume description	$08, three bytes, root descriptor LSN	00 00 18
report directory entry	$00–$1C, name field	72 65 70 6F 72 F4 → **report**. The high bit of its last character marks the end
Same 32-byte entry	$1D, three bytes, descriptor LSN	00 00 28
File descriptor at $28	$00, attributes, followed at $01 by two-byte owner ID	03 → owner read/write. Owner 00 00
File descriptor	$08, one byte, link count	01
File descriptor	$09, four bytes, file length	00 00 00 17 → 23 bytes

Structure	Offset and width	Value in the example
First segment entry	$10, three-byte start, then two-byte sector count	00 00 29 00 07 → seven sectors beginning at $29
File data	$00 from start of the first segment	65 63 68 6F → **echo**

A descriptor can hold more segment entries when a file grows in disjoint runs. Use the declared file length to stop. The descriptor's link count is a filesystem property. A module's in-memory link count has a different lifetime and purpose.

Identify an image before changing it

free /h1 reports the filesystem's volume name, capacity and free space. A copied disk keeps its volume name, so the name alone cannot identify the right backup. Use a known file or a time-specific marker, then confirm the Mac file assigned to the drive. Recover the right copy uses both checks.

For the layout and terminology, see The NitrOS-9 Project, *NitrOS-9 Technical Reference*, revision 0.1 (2004), chapter 5, pp. 48–52. The example sector numbers and file contents in this atlas were measured on the VCC Plus practice disk.

System calls and errors

Appendix C

Beacon takes the same route to the monitor and the disk. Its assembler emits a software interrupt instruction (SWI2) followed by one service byte. NitrOS-9 reads that byte, consults the process's registers, and dispatches the request. A successful write returns to the next instruction in Beacon. An exit ends the process.

The two calls in Beacon

Service	Before the call	After the call	Where you can see it
I$WritLn · $8C	A holds the output path number. X points to the first byte. Y holds the maximum count. The buffer contains a carriage return.	On success carry is clear and Y gives the count written. The line write honors the first carriage return within that limit. On failure carry is set and B identifies the error.	Follow the buffer through the assembler listing and change its destination.
F$Exit · $06	B holds the exit status. Beacon supplies zero after success or the error number returned by the write.	Control passes to the operating system. A waiting parent can receive the child's ID and status.	The shell accepts a new command after Beacon finishes. See the process lesson for the parent's view.

`I$` identifies an I/O request and `F$` a kernel service. Both use the OS-9 calling convention. The mnemonic after the prefix matters: I$Write takes a byte count and can transmit binary data containing carriage returns. I$WritLn has line semantics. For software that reads a line, I$ReadLn takes a path in A, buffer address in X and capacity in Y, and returns the actual count in Y on success. Keep the numeric path separate from the address of the character buffer.

I/O service finder

These are the named I/O requests in this image's system definitions. File managers implement operations appropriate to their devices. For example, seeking to a byte position makes sense on a disk file. A terminal has different status operations. The device chapter shows how OS-9 chooses the file manager.

Service	Request
I$Attach, I$Detach	Acquire or release an attachment to a named device. An open path can keep the device in use.
I$Dup	Get another path number referring to an existing open path.
I$Create, I$Open, I$Close	Create a new entry and open it, open an existing object in an access mode, or release an open path.
I$MakDir, I$ChgDir	Create a directory or change the calling process's working directory for data or execution.
I$Delete, I$DeletX	Delete using the data directory, or specify the mode used to resolve the pathname.
I$Seek	Move the position associated with a random-access file. The next transfer starts there.
I$Read, I$Write	Transfer a counted buffer. Check the returned count as well as carry.
I$ReadLn, I$WritLn	Transfer a line using a bounded buffer. The terminator matters.
I$GetStt, I$SetStt	Inquire about or change a path/device setting. A second function code determines the operation, its registers and whether the manager supports it.

Services encountered by a user program

Use this index to recognize names in source, disassembly and the guest's /DD/DEFS/os9.d. The definition file assigns numbers across several OS editions and extensions. A name in the file does not establish permission or implementation in the running kernel.

Look for	Services and their purpose
Start, replace or finish a process	F$Fork creates a child. F$Chain replaces the current program. F$Wait collects a child termination, and F$Exit supplies its status.
Process identity and scheduling	F$ID identifies the caller. F$SPrior sets an eligible process's priority. F$Sleep waits for ticks, F$SSpd suspends, F$Send delivers a signal, and F$Icpt installs its handler. F$SSWI sets a software-interrupt vector.
Find and release modules	F$Link searches by module name. F$Load reads a file and links a module. F$UnLink releases a reference and F$UnLoad looks up the module to release by name. F$NMLink and F$NMLoad are non-mapping variants.
Measure and request memory	F$Mem adjusts the caller's data area. F$GPrDsc, F$GBlkMp and F$GModDr request copies of process, block-map and module-directory information. F$CpyMem copies across an address-space boundary subject to the kernel's rules.
Process text, names and checksums	F$PErr writes a system error. F$PrsNam parses a pathname component and F$CmpNam compares names. F$CRC updates a cyclic redundancy check. F$SchBit, F$AllBit and F$DelBit search, allocate or clear bits in a map.
Clock and identity changes	F$Time reads the clock and F$STime sets it. F$Alarm and F$TimAlm concern alarms. F$TPS reports ticks per second. F$SUser changes a user identity subject to privilege checks.

Services used inside the operating system

System code also uses SWI2 services. A driver running in system state can manipulate data structures an ordinary application must reach through a public service. Encountering a name here is a reason to identify the caller's privilege and the exact kernel edition before adapting a listing.

Facility	Names to look up and what they connect
Interrupts and scheduling	F$VIRQ, F$IRQ and F$SSvc install service entries. F$IOQu, F$AProc and F$NProc manage I/O waits and runnable processes.
System storage and descriptors	F$SRqMem and F$SRtMem acquire and release system storage. F$Find64, F$All64 and F$Ret64 locate, allocate and release descriptor slots. F$AllPrc, F$DelPrc and F$GProcP concern process descriptors.
Module directory and boot	F$VModul validates a module. F$SLink, F$ELink and F$FModul link or locate directory entries. F$GCMDir compacts the directory. F$IODel releases an I/O module. F$Boot and F$BtMem serve bootstrap code.
RAM and process images	F$AllRAM, F$DelRAM and F$AlHRAM allocate or return RAM. F$AllImg, F$DelImg, F$SetImg, F$MapBlk and F$ClrBlk maintain process address images. F$FreeLB and F$FreeHB find free blocks.
Tasks and DAT translation	F$AllTsk, F$DelTsk, F$SetTsk, F$ResTsk and F$RelTsk maintain task mappings. F$DATLog converts a block and offset into a logical address. F$DATTmp is a historical temporary-image service marked obsolete in the definitions.

Facility	Names to look up and what they connect
Access across mappings	F$Move copies a block of bytes. F$LDAXY, F$LDAXYP, F$LDDDXY, F$LDABX and F$STABX load or store across task mappings. Examine the register contract of the particular entry before using it.
Edition- or hardware-dependent services	F$Debug and F$RegDmp concern debugging. F$ReBoot, F$CRCMod, F$XTime, F$VBlock and F$NVRAM concern reboot, CRC policy, extended time, validating a block of modules, or nonvolatile hardware storage. Treat these as names in the prepared definitions until the specific kernel and driver support is known.

Use an error as evidence

Guest result	What happened here	What you can do
211 · End of file	A reader requested more input after the writer finished or the file ended.	A program consuming a stream can treat this as its normal end condition. Distinguish it from an unexpected read failure.
214 · No permission	An open or operation requested access the target did not grant. The file lab removes read permission from a practice copy.	Check the path, owner, requested access and attributes. Restore the copy's owner-read permission.
216 · Path name not found	A path component failed to resolve. A procedure's data directory can differ from the directory containing the command program.	Inspect the current directory and try the full known data path.
218 · File already exists	Creation of a new output collided with a name already on disk.	List the existing file. Choose a fresh output name when you want to preserve both results.
228 · Process aborted	The parent reported the termination of the Sleep child deliberately signaled in this lab.	Check the process list again. Absence of *that child* and a responsive parent shell are the two expected observations.

Error 043 from RUNB is a language-runtime report. In the Night exercise the successful argument is the packed procedure's internal name, `night`, with the execution directory set to /H1. An error from a language tool need not be a kernel file error. For historical numbers and additional error families consult the bibliography in the language and edition guide.

Choose a language and a delivery format

Appendix D

Start with the program's intended home. A BASIC-hosted binary can expect a running BASIC environment. An OS-9 process receives an entry point, private working storage, paths and operating-system services. Choosing the environment tells you what to build, how to start it, and what a successful return looks like.

Match the artifact to its consumer

What you have	Who reads it	How this book used it
night.b09	BASIC09 in the OS-9 guest	LOAD reads the editable procedure source into the language workspace.
night	The guest's RUNB runtime	BASIC09 PACK produces a named procedure module containing intermediate code. RUNB takes the procedure's name.
beacon.a	The supplied OS-9 assembler	ASM creates a complete 6809 program module. The source contains the module, entry, storage and closing CRC directives.
beacon	OS-9's loader	Shell+ executes it by name or path. It receives the standard paths and returns by F$Exit.
watch.c	The period C compiler in the guest	CC drives several tools. RMA produces relocatable object code and RLINK joins it with the C startup object and library.
A CoCo BASIC program or BASIC-hosted machine-language binary	The corresponding ROM BASIC environment	Use the native BASIC, C and assembly books for their VCC Plus workspaces and direct-run conventions.

The output format is part of an assembler's job. For verification, the bundled LWASM was also run in its OS-9 module output mode on the unchanged Beacon source. Both guest-assembled files matched its 56 output bytes exactly. The reader can produce and execute the complete module using the guest alone.

Compile a small C program in NitrOS-9

This optional exercise uses the **cc version 2.3.0** present on the supplied HD0. Start at Shell+ with the usual **/dd/cmds** execution directory. Use a personal HD1 with fresh names **watch.c** and **watch**. Run:

```
chd /h1
build watch.c
```

Type the Watch source, one line per question-mark prompt, and press Return on an empty line to finish. This short program uses the C dialect accepted by the period compiler. Its untyped **main()** uses the compiler's implicit integer return type. The library routine **puts** writes a string and ends its line. A return value of zero tells the startup code the program completed successfully.

```
main()
{
    puts("night duty");
    return 0;
}
```

Confirm with **list watch.c** that the opening and closing braces appear on their own lines. In the guest keyboard matrix Control-comma produces the opening brace, and Control-period the closing brace. The Mac's natural keyboard layout also maps brace characters to those combinations.

```
cc watch.c -f=/h1/watch
```

Wait for the shell prompt. The explicit output filename is important. The source lives in the data directory, but the linker treats its final output as an executable. Giving it a full HD1 path keeps that file on your practice disk. The compiler created intermediate files as it worked and printed the stages it invoked.

Stage shown on screen	Job
c.prep	Preprocess the C source, including any requested headers or macro expansions.
c.pass1 and **c.pass2**	Translate the C program in successive passes.
c.opt	Optimize the generated assembly.
rma	Assemble a relocatable object whose final addresses can be resolved at link time.
rlink	Combine the object with **/DD/lib/cstart.r** and **/DD/lib/clib.l** to make the executable.

```
/h1/watch
ident /h1/watch
```

Look for **night duty** and a return to Shell+. Both guest processors produced a 1,759-byte module with a good CRC of **$F01AE3** for this exact source and toolchain. Your seven-instruction Beacon supplied its own entry and exit. This C program also carries the startup and library code needed by what you wrote.

```
?

(Term!02)/H1:list watch.c
main()
{
    puts("night duty");
    return 0;
}

(Term!02)/H1:cc watch.c -f=/h1/watch

cc version 2.3.0
'watch.c'
   c.prep:   watch.c
   c.pass1:   ./ctmp3.m -o=./ctmp3.a
   c.pass2:   ./ctmp3.a -o=./ctmp3.i
   c.opt :   ./ctmp3.i ./ctmp3.m
   rma   :   ./ctmp3.m -o=./ctmp3.r
   rlink :   /DD/lib/cstart.r ./ctmp3.r -o=/h1/watch -l=/DD/lib/clib.l

(Term!02)/H1:/h1/watch
night duty

(Term!02)/H1:
```

Figure 15: The running guest lists the C source, shows the compiler passes, assembler and linker, then executes the finished OS-9 command and prints night duty.

Read further

- Microware Systems Corporation. *BASIC09 Programming Language Reference Manual*, revision H (1984). See “Introduction to BASIC09 Programming” and “Program Statements and Structure” for procedures, loops and declarations.
- Microware Systems Corporation. *C Compiler User’s Guide for OS-9* (1983), chapter 1, “The C Compiler System.” The compiler front end and flags above were checked on the supplied image.
- Tandy Corporation. *OS-9 Level Two Development System* (1987), “Relocating Macro Assembler,” chapters 1–2, 5 and 9. Use its discussion of sections, external references and the linkage editor when you move beyond a single module.
- The NitrOS-9 Project. *NitrOS-9 Technical Reference*, revision 0.1 (2004), chapters 3–4 and 8. See the module header layout, standard paths and the system-call interface used in Beacon.

Find your way back into the system

Appendix E

An OS-9 manual from the 1980s can explain a process descriptor in today's emulated CoCo. Its example boot disk may give the terminal a different driver or start a different shell. Use the book to understand the structure, then use the guest to identify the particular parts you have.

Which edition is on the screen?

The prepared image identifies itself as **NitrOS-9/6809 Level 2 V3.3.0** and dates its build August 14, 2015. Its interactive shell identifies itself as **Shell+ v2.2a**. All required programs and practice files in these chapters use this supplied system and your own writable HD1. The guest operating system, shell, BASIC09 and C compiler each have their own versions.

Question	Where to look
Which processor and memory did I configure?	The running VCC Plus machine and its profile. The book's exercises use 512 KB. A 6809 program can also run on the HD6309 in its startup mode. Both runs here used the same 6809 OS image.
Which shell, disks and device drivers am I actually using?	Read the startup banner, check the two mounted hard-disk images in the Mac controls, then consult the device table.
Why does a command in a period manual behave differently?	Compare the manual's shell and utility editions to the command finder and the guest's own help. For example, the second greater-than sign in this Shell+ output syntax selects the error path.
How do I get back to a known boot?	Preserve the bootable system image and restore the personal practice data. For a separate boot experiment, start from a **fresh personal copy** of the built-in profile and confirm it reaches Shell+ before changing it. A working copy of HD0 together with its matching profile gives you a route back to that boot.

Read by question

- **I'm new to OS-9.** Start with Book 8's practice setup. Once you have recovered your captain's log, write the station procedure.
- **Where are my files?** Trace the file through the data and execution directories, then follow its sectors. The storage book explains why the same VHD also holds BASIC volumes.
- **What is the CPU actually doing?** Start with the CoCo 3 MMU lesson, then match a guest process to physical memory. The debugger workbench identifies the Mac controls used to inspect the machine.
- **I want to write a command.** Build the same kind of executable the shell knows how to start. Follow Night in BASIC09, Beacon in assembly, or the guest C build. Book 6 covers the Mac CMOC workbench for BASIC-hosted programs.

Trace a command lookup in a period guide

When you type a command name, where does the shell look after checking the module directory? Choose **Help → Archive Library…**, select **Entire Archive** and search for `OS-9 Level II module directory`. Under Dale Puckett and Peter Dibble's *Complete Rainbow Guide to OS-9 Level II*, open the result for PDF page 79. The

printed page number is 64. It explains how the shell looks in the current execution directory when the requested module is absent from memory. Compare the guide with the Echo module on your practice system.

Choose **Copy Citation** to keep the Archive's original ZIP link, the name of the PDF inside it and the PDF page you opened. If you refer someone to the printed book, include page 64 as well.

Books behind the exercises

- The NitrOS-9 Project. *NitrOS-9 Operating System User's Guide*, version 03.02.01 (2004). §§4.2–4.7, pp. 23–28, cover redirection, concurrent commands and procedures. Chapters 5–6, pp. 30–37, explain process scheduling and startup organization.
- The NitrOS-9 Project. *NitrOS-9 Technical Reference*, revision 0.1 (July 10, 2004). Chapters 1–4 describe the kernel, Level Two memory, module headers and paths. Chapters 5–7 describe the random block, sequential character and pipe file managers. Chapter 8 indexes the service requests. See F$Exit, p. 99, F$Fork, p. 100, F$Wait, p. 128, I$ReadLn, p. 144, and I$WritLn, p. 148.
- The NitrOS-9 Project. *The NitrOS-9 Level 2 Windowing System*, revision 0.1 (June 4, 2004), pp. 7–8, 20–23, 44, 49 and 58. These sections describe creating, selecting, drawing into and closing the graphics window.
- Tandy Corporation and Microware Systems Corporation. *OS-9 Commands Reference* (1986). Compare the original CHD/CHX, DIR, LIST, PWD/PXD and process-command entries with the contemporary guest's command help.
- Tandy Corporation. *OS-9 Level Two Development System* (1987). "Relocating Macro Assembler," chapters 1–2, 5 and 9. See the language guide for BASIC09 and C sources and the build paths used on this system.

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

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

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.

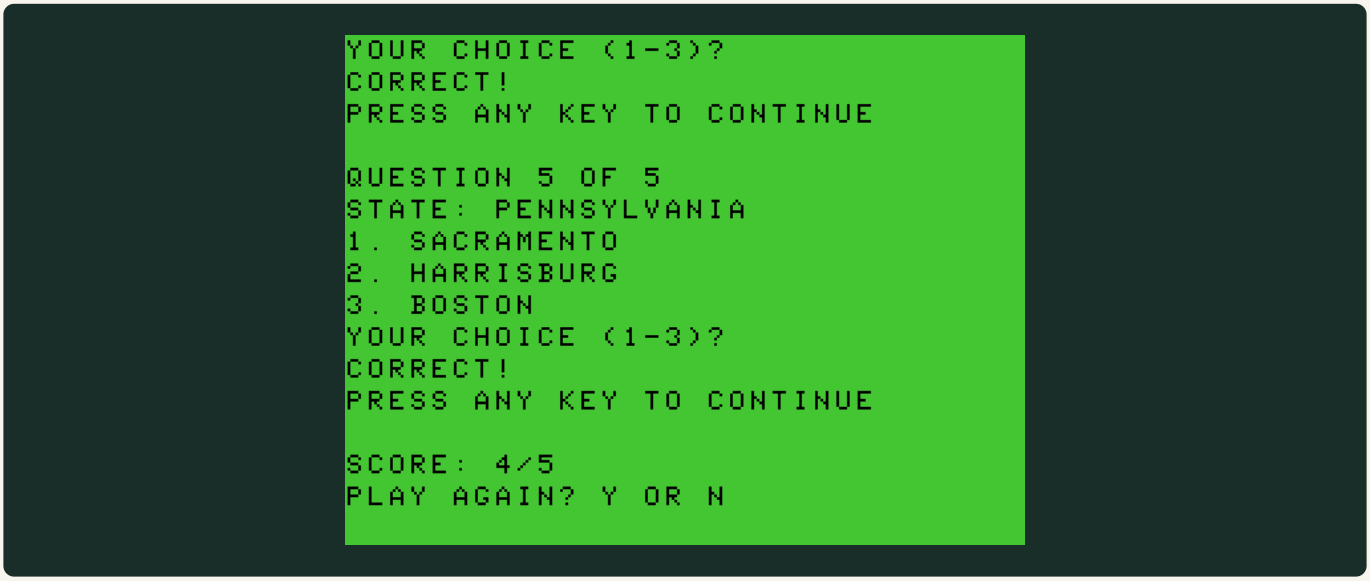
The CoCo 3 and CM-8 monitor presentation in VCC Plus's skeuomorphic System view was inspired by SockMaster's "pic of CoCo 3 and CM-8 monitor," published on CoCopedia. VCC Plus's artwork was designed and rendered independently. The remaining hardware illustrations were independently designed using historical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

Read this book online

CMOC license notice · LWTools COPYING notice

Vibe Coding with ChatGPT

A little Color Computer program can ask a surprisingly good question. Which planet would you like to visit? What is the capital of Kentucky? You can describe the game in ordinary words, ask ChatGPT to write a first draft, and watch a CoCo try it. The most interesting moment often comes when the CoCo does something neither of you expected.



Assembly quiz showing the correct answer to the fifth question, Pennsylvania, followed by SCORE 4/5 and PLAY AGAIN? Y OR N.

Brian K. Holman

Book 12 · VCC Plus 3.7.1

© Brian K. Holman 2026

Contents

Vibe Coding with ChatGPT 3
Make the first connection 4
From an idea to a working program 7
Give Mars a different greeting 10
Let the C compiler find the typo 12
Move a block one cell to the right 23
Build a quiz and give ChatGPT the keyboard 25
Find the first wrong result 33
Keep a field guide at hand 36
Start a project you can hand off 41
Publication and artwork notices 46

Vibe Coding with ChatGPT

In this book, *vibe coding* means making a program through conversation. You decide what the program should do. ChatGPT can read a draft, suggest a change and help investigate the result. VCC Plus gives you a place to see exactly what would change, decide whether to keep it and run it on the emulated machine. By the end you'll have a way to answer a very practical question. Did the program you kept actually do what you asked?

Set a place at the desk

You'll need VCC Plus for Mac with the build 30 interactive controls, the ChatGPT Mac desktop app signed in with access to **Work** and local MCP servers, and a folder for practice files. The lessons use a CoCo 3 running Disk BASIC. The C and assembly workspaces use the tools supplied with VCC Plus. Start with a disposable program, a few minutes at the keyboard and permission to share its contents with ChatGPT.

If you've never written a program, start with the first three chapters. One sentence changes a greeting, and you can try both answers. If you already program on a CoCo, make the first connection, then pick the chapter for your language. The capitals workshop brings the three paths back together.

The capitals workshop includes the complete example sources and a CoCo disk image. You can keep a copy to compare with your own first draft. In Chrome on Mac, Option-click a source link to save the file if a regular click opens it as text. Outcomes identified as **recorded** come from running the examples in VCC Plus. Prompts are invitations to try a task yourself. The words and code you get on your Mac can differ.

Other books on the shelf

Choose **Help → Manual Library...** in VCC Plus. Programming in CoCo BASIC on the Color Computer 3 is a good companion for numbered listings. Programming in C on the CoCo 3 — CMOC and Its APIs covers the C workspace and the bundled libraries. Your First 6809 Programs starts with registers and memory. Inside the CoCo 3 — Hardware and Debugging follows a running program more closely. You can also ask ChatGPT to find a section of an installed book during a lesson.

For historical magazines and manuals, the Archive Library has its own search.

Starting a larger project? Start a project you can hand off supplies an editable setup prompt and a downloadable folder of project standards, connection checks, test records and decision templates. The original CMOC and LWTools references cover the bundled compiler, assembler and linker alongside this library's teaching books.

Make the first connection

Chapter 1 of 9

On your Mac, two windows will share the work. The VCC Plus editor holds your program. The ChatGPT window is where you describe what you'd like to make. A small helper passes requests between them using the Model Context Protocol, usually called **MCP**. Let's give the helper one page to read and see whether it can find a message you left there.

Put a message in a practice draft

1. Copy **VCC Plus** from its disk image into **Applications**. Eject the image and open the installed app. Open the signed-in **ChatGPT** desktop app on the same Mac.
2. In VCC Plus, choose **Develop → Connect to ChatGPT...**. It opens the main BASIC draft and its connection panel. Look at which draft the panel names. If it contains work you care about, leave it alone. Open a separate practice draft in **BASIC Development** using **New**. A new editor can begin with the Earth or Mars starter.
3. In the practice editor, add a numbered remark near the top. For example, type `5 REM BLUE LANTERN` on a line by itself. **Make up your own two words** so the answer is something only this editor can supply. Keep the remark as the lowest numbered line. Save this practice listing as `Lantern.bas` in your own folder.
4. Open **Local Assistant Access...** from that editor's **...** menu. Verify the document named in the panel is the one you just prepared. Each BASIC connection shares a particular draft.

A BASIC remark is a note to a reader. You can leave one in the listing without changing how the game plays. You won't need to start the emulated CoCo for this first check.

Give ChatGPT a page to read

Choose **Save Read-Only Connection...**. Save the connection file in a private folder you'll recognize. It grants access to the selected document. Keep it out of the conversation and out of any folder you plan to share with your project.

Choose **Add to ChatGPT** in VCC Plus. Open a **new chat** in the ChatGPT Mac app and choose **Work** to use the local computer. Type `/mcp` if you'd like to see the configured servers. Seeing VCC Plus there is a good first sign. Now try a request that needs the actual editor.

> Use the VCC Plus BASIC connection. Read the current shared draft. Tell me its title and quote the lowest numbered line exactly.

Compare the reply with the line you typed. Did it find both of your words? If you chose `BLUE LANTERN`, for example, the answer should include `5 REM BLUE LANTERN`. You can also use **Copy First Prompt** in VCC Plus for its built-in connection check. Leave the shared editor and VCC Plus open while you work.

For a new project, **Copy Project Setup Prompt** provides a longer, editable starter with connection-setup permission, reference priorities and working rules. Choose **Edit...** to review it first. Copying changes only the clipboard; review the text before sending it to your assistant. See the project-starting chapter for the starter documents and the separate project, manual and Archive checks.

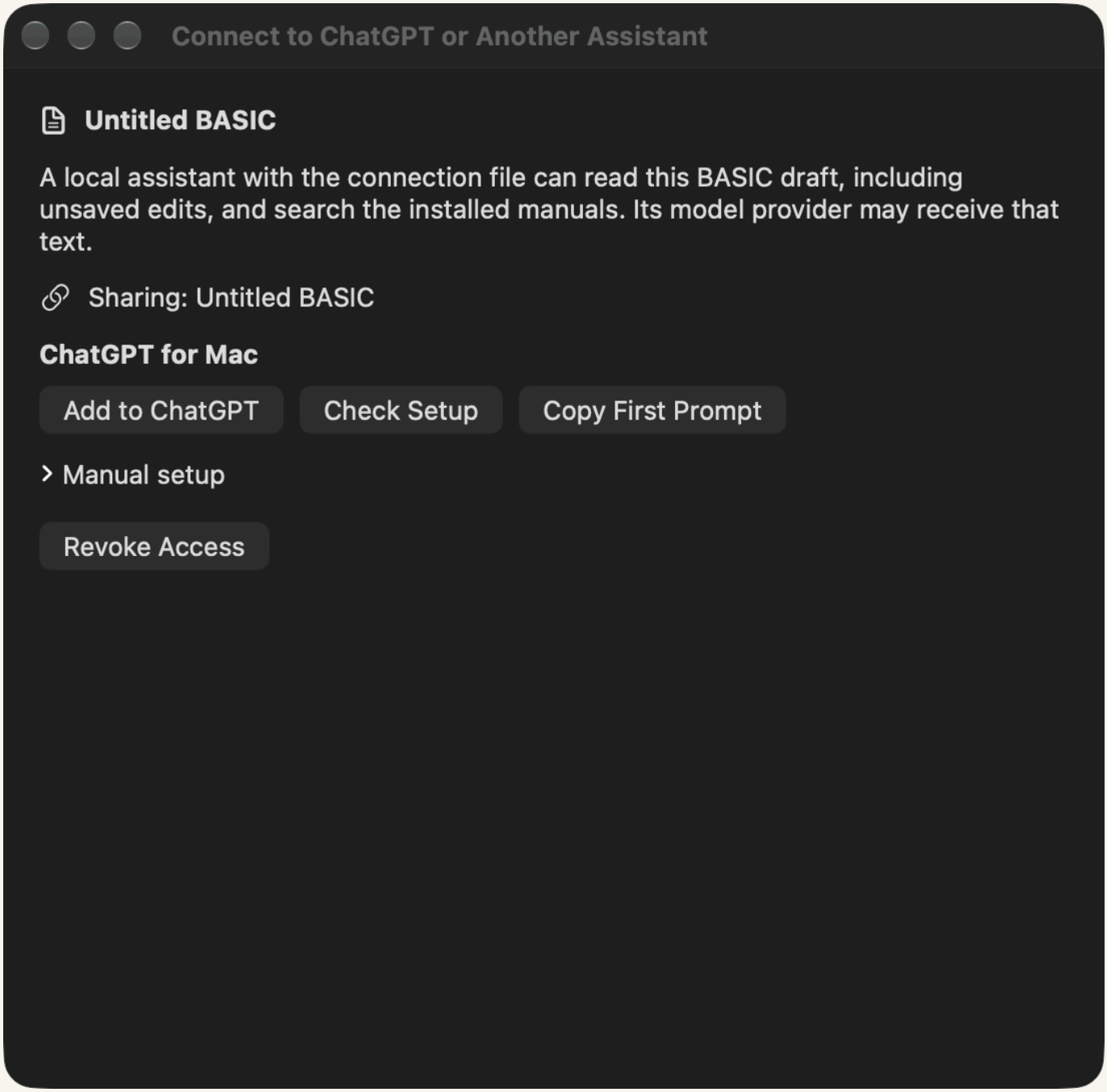


Figure 1: The BASIC connection panel identifies the shared draft above Add to ChatGPT, Check Setup and Copy First Prompt. Here the practice draft is named Untitled BASIC.

The connection to VCC Plus runs on your Mac. When ChatGPT reads a listing, diagnostic or screen through it, that text can go to the model provider and appear in the chat. A small practice project is a good first choice.

If the message doesn't arrive

- **ChatGPT guessed or gave you instructions for opening the file.** Make sure you're in **Work** in a new chat on the Mac. Ask it to use the VCC Plus tool to read the open draft and report the tool's response if it fails. Leave your secret words out of the prompt.

- **It quoted a different program.** Compare the named draft with the one in the connection panel. Open the right editor's **Local Assistant Access…**, save its connection, and add it. Name the intended workspace when more than one is available.
- **VCC Plus is missing from the list.** Use **Check Setup** in VCC Plus. If necessary, expand **Manual setup**. In a ChatGPT version offering **Settings → MCP servers**, choose **Add server → STDIO**. Copy the displayed name, command and two arguments into their respective fields. Each argument gets its own entry. Save, restart that server and repeat the reading test. This walkthrough uses a ChatGPT account and workspace that offer local servers on the Mac. OpenAI's desktop MCP instructions

Let the assistant propose an edit

Return to the practice draft's connection panel. Select **Allow proposed edits and controls for runs I start** and choose **Save Review and Run Connection…**. Use the updated connection in ChatGPT. If you replace the file used by an existing server, restart that server to make it read the new grant. Repeat the lowest-line check. For interactive lessons, save a fresh connection from the installed app and check its advertised controls.

You still have two decisions to make in VCC Plus. First you can accept or cancel the source change. Then you can start a shared run. Once a supported run starts, ChatGPT can read its text screen and send supported keys to that run. The C workspace also has a separate, explicit permission for disposable playtests; that grant permits repeated artifact-bound test launches without another launch prompt.

Use a separate connection file for each workspace you want to keep available. The equivalent panel is **Assistant…** in C Development and **Local Assistant Access…** in Assembly Development. All three workspaces use the same helper. A BASIC connection points to one draft. A C or assembly connection points to one project.

When you finish, choose **Revoke Access** in the workspace's panel to disable that grant and remove its file. Previously shared results can still appear in the chat. To work together again, save a new connection, reconnect and repeat the reading check.

An existing local Codex chat

The installed helper also works with a local Codex chat. Add its STDIO command and the two separate grant arguments in **Settings → MCP servers**, restart the server, then use `/mcp` in the existing project chat to check live discovery. Keep VCC Plus and the shared C project open. Use the private connection exported for that project, and keep it outside Git. Official OpenAI MCP setup

For a custom Make project, select **Use Project Makefile** in the C project's Assistant panel before exporting. The C workflow explains the build-support folders and configuration choices.

Check discovery separately from access

A saved server, tools visible in this chat, and a successful call are three different checks. Ask the assistant to discover its tools, call `get_capabilities`, and read the actual shared title and revision. For an updated C connection, also ask which capabilities it advertises: project refresh, diagnostics, disposable playtests, audio and processor selection. Installing a new app does not silently expand an older private grant.

If the server is registered but its tools are missing, use the client's supported **Restart** action and check its catalog again. A delayed catalog is not a reason to grant more files or keep exporting new connections. If the reply instead says `host_unavailable`, leave VCC Plus and the intended project open and check that the saved grant is still valid. That failure is different from a missing catalog.

For a technical connection check, the same installed helper and approved private grant support ordinary STDIO MCP: `initialize`, `notifications/initialized`, `tools/list`, then an advertised `tools/call`. Ask the assistant to report that result separately from tools visible in the chat. It must keep grant contents private and use the public helper; reading app caches or calling a private emulator bridge is not part of this workflow.

From an idea to a working program

Chapter 2 of 9

Suppose you'd like a scorekeeper for a game at your kitchen table. “Make me a scorekeeper” leaves ChatGPT to invent the rules. Pick two keys and write down what you expect them to do. Now you have both a request and a way to judge the first answer.

Describe a turn you could play

In a separate practice BASIC draft with review access, try the request below. Starting a practice draft and connecting it are covered in Chapter 1. You can also use this as a pattern when you design the capitals quiz.

> Make a small CoCo 3 Disk BASIC scorekeeper in this draft. Start at zero. Show which key to press. The 1 key adds one point. The 2 key prints FINAL SCORE followed by the total, then ends. Other keys leave the total alone. Read one key at a time. Propose the numbered listing in VCC Plus and wait for my review.

Prediction. If you send `1`, `1`, `0`, then `2`, the final score should be 2. Jot down that prediction before you read the proposed solution. It gives you a test that stays the same even if the program changes.

If you want to understand the keyboard routine, ask for a reference while you work.

> Search the installed VCC Plus manuals for INKEY$. Read a matching BASIC section. Give me the book and section title and explain what the program gets when nobody is pressing a key.

All three workspaces can search the installed manuals. ChatGPT can use `search_help` to find a section and `read_topic` to read it. You can follow the answer in **Help → Manual Library...** with **Ask the Manuals**. For this program, an empty string means no key is waiting. Polling again keeps the scorekeeper ready for your next turn.

VCC Plus also has a separate read-only MCP connection for historical publications. Open **Help → Archive Library...**, select the link button, choose **Save Archive Search Connection...**, then **Add to ChatGPT**. In a new Work chat, try “Search the Color Computer Archive for a BASIC manual page about INKEY$. Give me the manual title, the physical PDF page and the original Archive link so I can check it.” You can open that page in the Archive Library and compare its explanation with the behavior of your scorekeeper.

If the reader offers **Copy Snippet** or **Open in BASIC Editor**, it has a reviewed listing associated with that page. Keep its citation when discussing or adapting the example. Opening a BASIC listing creates a separate editable copy; inspect it before choosing Run. A photographed program or an OCR search hit does not by itself promise corrected, runnable source. The handbook's Archive reader walkthrough explains the page, citation and listing controls. The Archive connection remains read-only; use your separate development connection to propose project edits.

When you are researching software already in Software Library, look for **Guides & References** in its details. A reviewed walkthrough, map or tutorial opens directly at its Archive document/page. These links supplement the main manual; they do not assert that every title has been researched exhaustively. Keep the historical citation when it informs your project. Both libraries resume their browsing/reading contexts, and the Manual Library's **Bookmarks** list can keep useful native-book and toolchain-reference sections together. Those personal reading choices do not add MCP permissions or expose your private library.

Read the change before you play

Look at the proposal in **VCC Plus**. For a whole BASIC listing, check the first and last lines and then follow the short path that matters. Where does the score start? Which line adds one? What takes the player back to the key check after an invalid choice? Ask ChatGPT to point out the numbered lines if you'd like help reading them.
Choose **Apply Edit** when the proposal matches the game. A code block posted in the chat still needs to be proposed through the connection before you can review it in the app.
For a C or assembly project, you can make the same request in smaller pieces. Tell ChatGPT which existing files to change. Read the proposed changes together, apply them, then ask it to build the new revision. The compiler or assembler can catch a misspelling before you even start the CoCo. A successful build is your cue to try the game.

Start the CoCo and watch one turn

Ask ChatGPT, "Check the machine status for this draft. Tell me whether I can run it and name any blocker you find." Use a disposable Disk BASIC session. Save anything you want to keep before replacing a guest program. If the app reports writable media, protect or eject the disk after saving your work. Eject the starter cassette, finish any recording or Mac shell session and turn off DriveWire networking if the readiness check calls for it. The CoCo should be running at its supported BASIC prompt. Chapter 8 explains how to read a refusal.
Choose **Run** for the applied BASIC proposal in VCC Plus. For C and assembly, start the current build from the shared project as described in Chapters 4 and 5. Back in ChatGPT, ask for this test.

> Use the run I just started. Read its CoCo text screen. Send 1, 1, 0, then 2 through the VCC Plus game controls, one key per call. After each key wait for the program to accept it before continuing. Quote the final score from the CoCo. Tell me whether it agrees with my prediction of 2.

A choice game is easiest to test when it shows that it heard each key. Perhaps you see the updated total after a point and a short reminder after an invalid choice. If a screen stays the same, ask what the assistant actually observed. Avoid repeating a key until you know what happened. You can request visible feedback as a refinement.

Change one rule and keep the evidence

Suppose the score is right but your first-time player doesn't know how to quit. Try, "Add a line at startup that says PRESS 2 TO FINISH. Keep the rules and the scoring the same. Propose the smallest change that does that." Read the change in VCC Plus and run the same four keys again. Test an immediate 2 in a separate run as well. The total should be zero.
Save the source you want to keep from the native editor or project. Keep a short text note beside it. This pattern works for a ten-line game as well as a larger project.

Write down	What to put there
First request	The words you gave the model, the language and the target CoCo.
First attempt	Whether the first proposal was accepted, whether it built if needed, and what appeared on the first run.
A failed check	The exact input and the observed screen line or compiler diagnostic.
Refinement	What you asked for next, what changed, and which version you tested.
Working copy	The saved filename, target processor where relevant, and the last test someone else can repeat.

If the first run failed, keep that entry even after a repair succeeds. It answers a different question from whether the final game works. For a worked example with three first drafts and three recorded playtests, turn to the capitals quiz.

Give Mars a different greeting

Chapter 3 of 9

The CoCo asks you to choose Earth or Mars. Earth gets a blue greeting. Mars gets a red one. This time you'd like a visitor to Mars to see **Bring a warm coat!** You can describe that change to ChatGPT in a sentence. The interesting question is how you'll know the sentence made it all the way to the emulated CoCo.

Find the greeting

1. Open **Develop → BASIC Development**. If you have a draft you want to keep, save it. Choose **New** to open the Earth or Mars starter. In VCC Plus, the BASIC editor holds the listing on your Mac. The CoCo has its own program memory.
2. In this editor's **…** menu, choose **Local Assistant Access…**. Select **Allow proposed edits and controls for runs I start**, then **Save Review and Run Connection…**. Use **Add to ChatGPT** and open a new **Work** chat in ChatGPT for Mac. If you already connected this exact draft with review access, you can keep using that chat.
3. Ask ChatGPT to read the listing through VCC Plus. Give it a question whose answer you can check in the editor.

> Find the numbered line that sets the Mars greeting. Quote that line from my shared BASIC draft. What does each side of the colon do?

You should find the following line in both places.

```
90 PALETTE 6,37:M$="Hello Cold Red Dot World!"
```

BASIC executes two statements on line 90. The first assigns color 37 to palette entry 6. The second puts a string into the variable `M$`. A later `PRINT` displays the string. Changing the words inside the quotes lets you leave the palette and the keyboard routine alone.

Review one line

> Propose a one-line edit through VCC Plus. On line 90 change the Mars greeting to exactly "Bring a warm coat!" Keep the rest of the listing as it is.

Look in the VCC Plus review window. If ChatGPT only displays code in the conversation, ask it to submit the edit to the shared draft for review. Check the line number, the palette values, the quotation marks and the exclamation mark. Here is the line to look for.

```
90 PALETTE 6,37:M$="Bring a warm coat!"
```

Choose **Apply Edit**. The new line now appears in the Mac editor. You can still choose **Undo Edit** at this revision. If the preview changes anything else, choose **Cancel Edit** and ask for the single line again. If you typed in the editor while the review was open, have ChatGPT reread the current draft and submit a fresh proposal.

Try the two planets

Use a running Disk BASIC system at its **OK** prompt. Save any guest program you want to keep before replacing it. In the review window, choose **Run Both Choices**. This button recognizes the starter with the exact revised line above. For a different listing, use **Run** and test the choices yourself or ask the connected assistant to press the keys.

Watch the results for each planet. VCC Plus tries **3** first and checks that the menu stays up. It then tries **2** and reads **Bring a warm coat!** from the guest. On a fresh run it tries **1** and reads **Hello Nice Blue Dot World!** The check samples the background color of each greeting. It also types `PRINT 7` after each run and looks for `7` followed by `OK`. That last exchange tells you BASIC is accepting commands again.

> Read the verification results from VCC Plus. What happened after key 3? Give the greeting and the observed background for each valid key.

Use **RGB** output for the starter's blue and red. On composite output, the Mars palette value can appear amber. If a run is blocked, ask which condition VCC Plus reports. A mounted cassette, a writable disk or active DriveWire can prevent a shared run. Use the native media controls to resolve the reported condition after saving any changes you want to keep.

Start with an empty listing

Save your planet program. Choose **Revoke Access** for that connection. Choose **New**, click in the source editor, select all its text and delete it. Share this empty draft with review access and give ChatGPT the next job.

> Propose a complete numbered BASIC listing for this empty draft. Print the squares of the whole numbers 1 through 4, one result per line, then end. Use VCC Plus to submit it for review.

Read the proposed loop before choosing **Apply Edit**. Which line repeats, and what ends the loop? Choose **Run**, then ask ChatGPT to read back the screen. Look for `1`, `4`, `9` and `16` in order. Change the request to stop at 5, predict the last result, and check it against a second run. Save the listing you want to keep.

Let the C compiler find the typo

Chapter 4 of 9

One extra letter can stop a whole C project from building. You'll put that letter in a practice copy of **Hello Disk**, ask ChatGPT to get the compiler's diagnosis and review its repair. When the build succeeds, the CoCo will read five words from a floppy. That gives you something to check beyond the message that says the compiler finished.

Give the compiler a small problem

1. Choose **Develop → C Development...**. Click **New**, choose **Hello Disk**, and create a practice project in a new folder. Open `main.c` and look near the end of `main` for the line that prints the generated word count.
2. Add a lowercase **s** to the end of the variable name. In the supplied project that's line 14. Leave the printed message unchanged. Check your altered statement against the one below.

```
    printf("GENERATED WORD COUNT: %u\n", generatedWordCounts);
```

3. Click **Assistant...**. Expand **Files shared**. Alongside `main.c` you'll see the header `include/hello.h`, the C files in `src`, and `assets/words.txt`. Choose **Save Private Connection...**, give the C connection its own filename, then use **Add to ChatGPT**. Open a new Work chat and use **Copy First Prompt** to confirm the project. The compiler can work with your open draft, including the typo.

> Build the current revision of my shared Hello Disk project. Report the first compiler diagnostic with its file, line and exact message. Read the relevant project header, then propose the smallest repair through VCC Plus.

CMOC reports `undeclared identifier 'generatedWordCounts'` in `main.c`. An *identifier* is a name in the source. Compare the diagnostic with the declaration in the header.

```
extern const unsigned char generatedWordCount;
```

C treats the singular and plural spellings as different names. The declaration tells the compiler that the project supplies an unsigned byte named `generatedWordCount`. Removing the final **s** makes the statement refer to that byte. If the initial build succeeds, compare the variable in your editor with the altered line above and ask ChatGPT which project and revision it built.

Apply, rebuild and read the disk

Read the current and proposed lines in the VCC Plus review window. Choose **Apply Repair** when the proposed change removes the extra letter. Look for the **applied revision**. A revision identifies the particular draft the assistant built or edited. The failed build belongs to the earlier revision. Click **Rebuild** to compile the applied revision and watch for the successful result.

Before starting the program, leave a running Disk BASIC system at an empty **OK** prompt. Hello Disk needs **FD0** empty so VCC Plus can temporarily mount its generated disk there. Save any disk changes before ejecting it. If a BASIC listing is still in guest memory from the previous lesson, save any open BASIC editor draft, use **Read from CoCo** and **Save As...** to keep the guest listing. When you are ready to clear its program and variables, type `NEW` at the CoCo prompt.

Choose **Run and Verify**. VCC Plus matches an instruction in the running binary to the source, continues execution, checks for a floppy read and shows the program's output. Look for **HELLO**, **WORLD**, **CMOC**, **VCC** and

PLUS, followed by **DISK RESULT: 0** and **GENERATED WORD COUNT: 5**. Compare the words with `assets/words.txt`. Save your repaired project if you want the corrected sources on disk.

Play an ordinary C program

For a program that takes keyboard input, create a separate project using **New → Hello World**. This starter offers Earth or Mars in a single C file. Share the new project from **Assistant…** and ask ChatGPT to build its current revision. An ordinary shared build compiles the project's C files with bundled CMOC. For a project with its own Make rules, use the connection described below. An ordinary shared build can run with a write-protected disk in FD0.

Choose **Run C Build…**. **Play in CoCo** continues from the first verified statement in `main` and lets the connected assistant read the screen and send game keys. **Inspect at main** pauses at that statement so you can ask which source line and machine address were reached. Choose **Play in CoCo** for this exercise and give ChatGPT the keyboard task below.

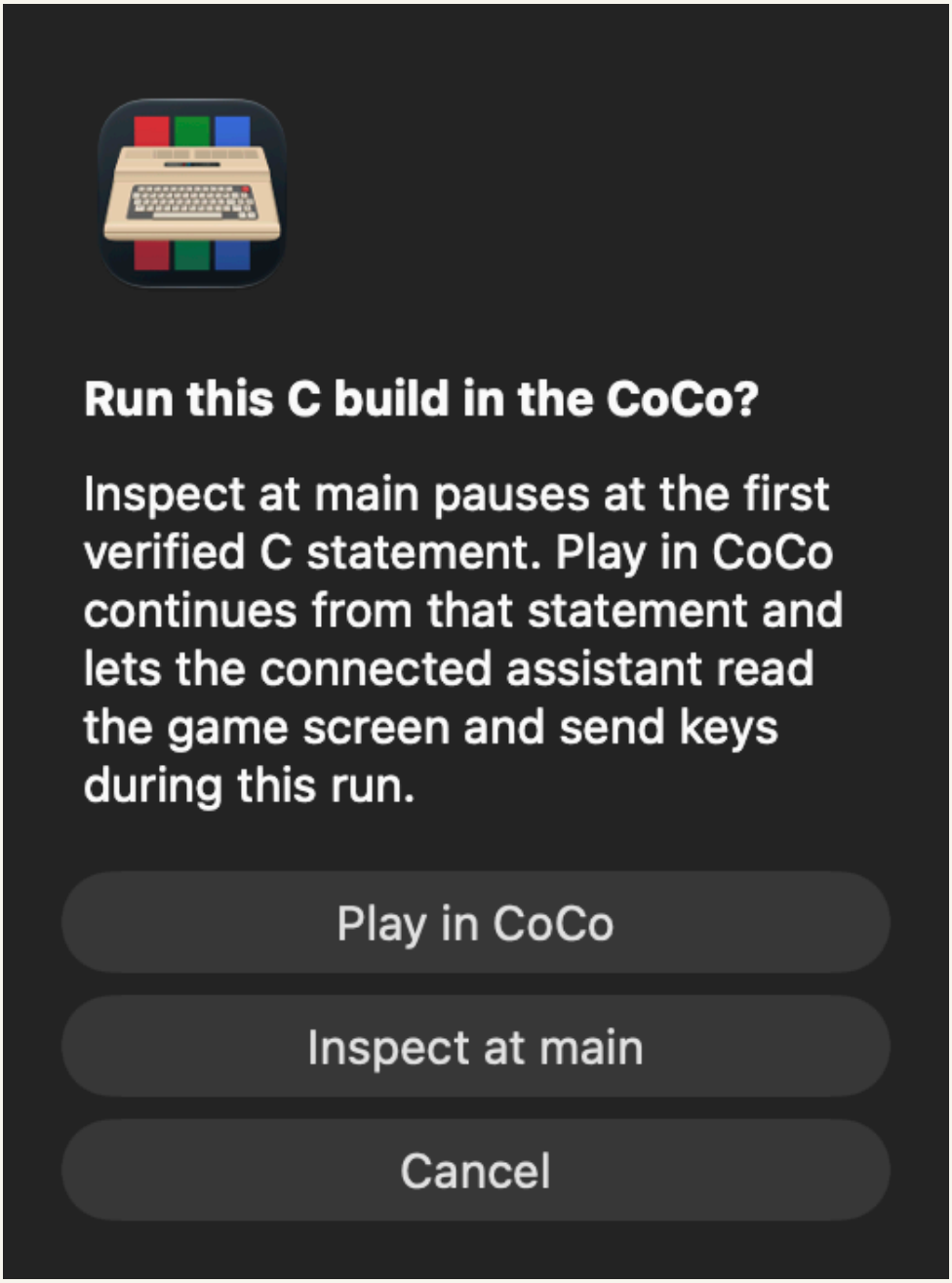


Figure 2: The Run C Build dialog offers Play in CoCo for the interactive game and Inspect at main for a debugger stop.

> Read the menu from this C run. Send the key for Mars, wait for the program to return to BASIC, and report the greeting from the final text screen.

Look for **Hello Cold Red Dot World!** The previous chapter changed the BASIC draft. This greeting comes from the C program you just built. A finished run can report the text captured at return. To choose Earth, start a fresh run.

Keep a project's Make rules

A larger game may combine C, inline assembly, generated graphics and a custom linker layout. Open its project folder in C Development. In **Assistant...**, select **Use Project Makefile**, check **Files shared**, and save a fresh

private connection. This permits the local host commands in that Make recipe and its compiler inputs. The native **Build** button and the assistant then use the project's declared recipe with the installed bundled toolchain. Its headers and inline assembly can stay in place.

Leave the checkbox off for the ordinary restricted C connection used earlier in this chapter. That builds ordinary sources with an app-generated recipe. Hello Disk also has its own installed recipe and verification flow. Choose the mode that matches the project you intend to share.

Suppose your repository contains `game/Makefile` and a sibling `tools` folder of packers. Open `game` as the project. In its existing `vccplus-project.json`, add `"buildSupportDirectories": ["../tools"]`, or enter `../tools` in Project Settings' **Build-support folders** field. The build snapshot copies that explicitly named folder beside the project so the recipe's `../tools` paths still work. Inspect the listed support folders before saving the connection.

> Identify this project's current revision, approved Make recipe, build actions and configurations. Build the current revision using its declared disk action and Release configuration, if those choices are available. Report the build ID, result and first diagnostic. Do not start the CoCo yet.

The assistant reads these choices through `get_capabilities` and requests the declared build or disk action through `build_project`. It can choose a listed configuration; it cannot invent a Make target or send extra command arguments. Source edits advance the revision. Assistant source-edit proposals do not change Makefiles, scripts or project configuration. Make those changes locally, then use the scoped refresh below and review the changed build controls. An older connection without refresh needs a fresh export instead.

A successful disk build does not establish that a packed game has a verified C run or debugger stop. A custom project can generate a banked debug manifest that binds its packed binary and disk to original source, linker, object and listing evidence. Its source set includes the complete retained compiler output, so large type records can be recovered from hash-verified assembly when the assembler listing truncates a long comment. VCC Plus checks the unpacked module bytes and actual physical RAM banks before showing source locations or values. Expected bank switches retain that connection. Optimized locals and unknown frames remain unavailable. A successful recipe is not proof of a debugger stop or of the game's behavior.

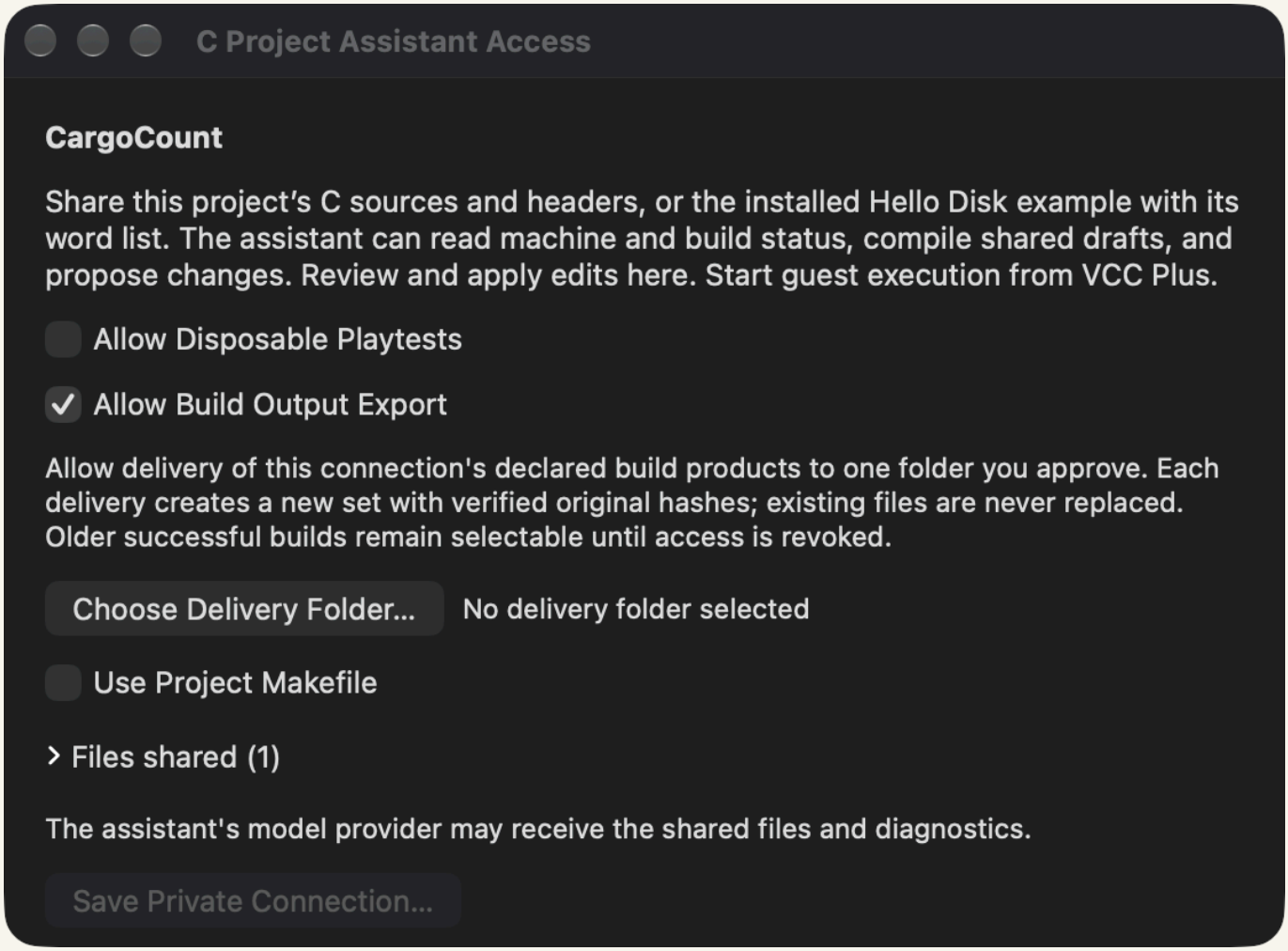


Figure 3: C Project Assistant Access offers separate choices for disposable playtests, build-output export and the project Makefile. Export is selected here, but no delivery folder has been approved, so Save Private Connection remains unavailable. Review the files and permissions before saving.

Deliver the build you already tested

A successful build has its own identity and frozen products. To let an assistant deliver those files, open **Assistant...**, select **Allow Build Output Export**, and use **Choose Delivery Folder...** before saving a fresh private connection. This permission is separate from permission to run disposable playtests. The assistant receives an opaque destination ID rather than access to browse Mac folders. Older connections keep their existing permissions.

> Build the normal Release and Debug configurations. Keep both build receipts. Export each configuration's declared binary, Home disk, Data disk and Home alias, if this project declares them, to the approved delivery folder. Do not build again to retrieve files. Independently compare every delivered length and SHA-256 with the original build receipt, then give me the two delivery sets.

`get_build` lists `exportable_outputs`. The assistant selects those output IDs and the exact original project/build/artifact/revision with `export_build_outputs`, then polls `get_build_export` until completion. Building Debug does not erase the previous Release choice. Each delivery gets its own verified folder, so identical filenames do not silently replace another configuration's files. An identical request retry verifies the prior delivery; a changed request or changed delivered file is reported as a conflict.

The binary, primary disk, existing debug manifest and additional playtest disks are declared products. Use `exportOutputs` in project settings for other concrete relative deliverables or aliases; an output directory is not a permission to export everything found there. A delivery preserves frozen build bytes. It does not include a playtest's later saves or publish software to the website. **Build → Show Build Products** remains available when you want the native route.

Refresh the inputs you meant to share

Connections that advertise `project_refresh` can inspect the approved project inputs and request a scoped refresh after an external edit. Ordinary edits to shared files update the revision. New shared files or changed build controls open a native review showing the additions and exact before/after contents. You can approve or reject; neither choice runs the project. Your unsaved drafts stay intact, and conflicting disk edits still need local resolution. Saving screenshots or receipts in a declared output folder does not change the source revision.

> Use read_project_inputs to list this project's included files and excluded output folders. I added a header locally. Request refresh_project at the current revision and report the review ID. Poll get_project_refresh for my decision, then identify the new revision before building. Keep unsaved drafts.

The manifest explains each input's role, inclusion and hash; it is paged, so ask for later pages when needed. New files, changed recipes and support scope need local review. Pending or rejected changes block builds and runs. Approval adopts the reviewed input set; it does not execute Make or start a game. If a file on disk conflicts with an unsaved editor draft, resolve that conflict locally.

Keep source resources separate from generated results in `vccplus-project.json`. `resourceInputs` names intentional data files, `generatedInputs` names retained generated inputs, and `outputDirectories` names result folders to exclude. These are literal project-relative paths. An approved sibling support folder remains an explicit choice; a dependency file cannot grant access to another folder. Touching an unchanged file alone does not change its content-based revision.

The build cache is a disposable working copy. VCC Plus can repopulate it when its receipts or dependencies no longer match approved inputs; it does not run an arbitrary project's `clean` recipe to repair the cache. A stale-state reply can name changed paths and expected/current identities. Read that evidence and refresh the actual project rather than deleting source or changing a request until a refusal disappears.

Let an assistant play a test copy

Select **Allow Disposable Playtests** in the C Assistant panel before saving a connection. This permits the assistant to launch the current built disk in a separate 512 KB CoCo, with MC6809 as the default and HD6309 available when the connection advertises processor selection, without changing your open machine or mounted media. It can hold combinations of physical keys and standard joystick axes and buttons for bounded video fields, capture the game's graphics, and cold restart its own writable test disk. You need not approve each test launch after granting this option. Rebuilding, changing the shared revision or revoking access ends that run.

> Build this revision's disk. Start a disposable playtest of that exact artifact. Capture its title screen, describe the controls, and test a short movement and jump. Report the run ID, field counts and what the images actually show.

The assistant uses `start_playtest`, `playtest_input` and `capture_playtest`. `restart_playtest` keeps the same writable disk, so save and load tests use the game's real controls. `stop_playtest` retains the test disk in VCC Plus

development results. Inputs have request IDs: an identical retry returns the prior receipt without repeating movement. The clock measures completed emulated video fields, not wall time or the game's own update count. A breakpoint can end an input plan early; read the returned clock and remaining field/restart allowance before continuing.

Starting a disposable playtest can return **starting** while VCC Plus verifies the build evidence and boots the machine. The assistant should poll `get_run` for progress and readiness before sending keys or debugging. Retrying the same request recovers that startup; a new request must not create a duplicate machine. The reserved run can be stopped while preparation is pending.

Keep the original project, revision, artifact and request IDs if a connection times out. `get_run` reports the preparation phase, `busy` and a terminal failure when one occurs. A ready startup has a worker `receipt`; check its actual state and stop evidence before claiming the requested function was reached. To cancel a pending start, use `stop_playtest`, then poll until cancellation has finished. An interrupted transport is not proof that no guest was created.

Typed answers can combine letters, `space`, `period`, `backspace` and `enter`. For example, the assistant can type **ST. PAUL**, correcting a letter with Backspace before submitting. These are physical CoCo keys; an empty key array releases them, and Shift works as a normal held chord. This applies to disposable playtests. The ordinary shared-run `send_keys` tool retains its smaller key set.

Test disk changes without restarting the program

A game can keep its program in memory while asking for another disk, or use Home and Data together in different drives. In **Project Settings → Build**, declare additional project-relative **playtest disk outputs**. The approved Make recipe must produce them with the primary disk; each run gets separate writable copies. Up to eight additional floppy products are supported.

Fresh multi-drive connections can select FD0–FD3. Earlier media connections keep FD0-only exchange, so inspect live capabilities after reviewing a new connection and restarting its helper. Omitted `drive` still means FD0.

> Build Home, Data and the deliberately wrong test disk. Start the owned Home run, pause at an appropriate prompt and insert the listed Data copy in FD1. Use the game's normal controls to load an uncached level, save to Home0 and reload. Compare per-drive activity and exact hashes: Data1 should remain unchanged when the game saves only to Home. Repeat the game's empty, wrong, damaged and write-protected media recovery paths without changing its memory.

`list_playtest_media` lists artifact products and their IDs/hashes. Include `run_id` to inspect that run's current private copies, because guest writes can change their hashes. `exchange_playtest_media` takes a listed media ID, its exact current hash and the target `drive`. On insert, optional `write_protected: true` provides a real protected private image; omission preserves its existing setting. The assistant supplies no Mac path. A writable copy cannot occupy two drives at once: eject it before moving it. Distinct private copies can have identical bytes.

Pause explicitly before inserting or ejecting. The shared floppy controller must also be idle: a keyboard prompt or stopped motor alone does not prove a pending command is complete, even if it belongs to a different drive. Replacement durably retains the outgoing disk first and affects only the selected drive. It does not reset the guest, change BASIC's default drive, alter RAM/registers or advance the field clock. Resume with normal input and inspect the game's response; a mount alone does not prove it accepted the volume, edition or checksum.

Fresh receipts show all four drives, their current copy identities, original and current hashes, protection, generation and actual transferred-byte activity. Read/write bytes are not counts of successful Save commands. Combine them with exact disk contents and the game's own save/reload result. Older top-level disk fields continue to describe FD0.

`restart_playtest` preserves all current drive assignments and writable copies; ensure FD0 holds the intended DOS boot disk first. It does not move Home from a nonzero drive automatically. `stop_playtest` freezes and

independently retains every disk, including ejected copies, before ending the worker. Later starts can select saved copies through `retained_media`; the primary retained-disk fields still identify the primary product wherever it was mounted. Saved originals stay unchanged. This resumes disk contents, not CPU/RAM state. Startup drive maps are not added: use explicit paused exchanges for the other drives.

Reuse a request ID only for the identical drive, selection and protection plan. After an exchange, obtain a fresh debugger snapshot even though guest time did not advance. A lost reply is uncertain; retry the identical request under current authority rather than issuing a new mutation that may repeat a completed swap.

Test a joystick without changing your controller setup

A fresh disposable-playtest connection supports both standard CoCo joystick ports, independent of your personal gamepad, mouse or keyboard assignments. For each `playtest_input` step, keep `keys` and `fields` and optionally add `joysticks`. Each port snapshot names `right` or `left`, integer `x` and `y` positions from 0 to 16383, and `buttons` containing `fire1`, `fire2`, both or neither. Center is 8191; X increases rightward and Y downward.

```
{"run_id":"<owned run>","request_id":"walk-left-fire","steps":[{"keys":
[],"fields":12,"joysticks":[{"port":"right","x":0,"y":8191,"buttons":["fire1"]}]},{"keys":
[],"fields":6}]}
```

Each step replaces the entire keyboard and both-port state. An omitted port centers and releases its buttons. Omitting `joysticks` centers both ports. Every completed or interrupted input plan releases keys and centers the sticks. The receipt's `input_trace` shows the input applied during each step and its actual completed fields; final `input` shows the released state. These observations confirm delivery, not that the game responded correctly. Read the game's controls and inspect screenshots of actual movement, jumps and menus.

`get_playtest_input` reads the owned machine's input and clock without advancing it. For the personal shared machine, `get_machine_status.input` instead separates the native controller preview, the last submission and the actual core input. It reports source, interface, connection and focus availability without revealing controller identities or unrelated key contents. Preview movement alone does not prove delivery; a paused machine releases its input. These tools do not alter personal input settings. Standard analog playtest input does not simulate the app's optional high-resolution joystick interfaces.

> Discover joystick support, then test right-port movement, Fire 1 and release through the guest's normal controls. Capture before and after frames. Compare the input trace with what the game actually did. Do not substitute keyboard arrows for a joystick-only acceptance test.

Compare two graphics pages without advancing

For graphics projects whose verified manifest declares extra buffers, pause and use `capture_playtest_graphics`. The assistant can compare the register-selected picture with one declared buffer by ID at the same stopped state. The returned images and physical identities help diagnose page flipping without writing RAM or advancing the guest. Normal `capture_playtest` still shows the last published video frame, which may precede a register change.

For a custom scrolling GIME game, the project can declare bounded `current_gime` canvases in its verified manifest. Captures preserve the live viewport, row stride, pan, palette and relative video offset when comparing the two pages. Fixed HSCREEN previews keep their BASIC layout. An escaping viewport is refused; the assistant must not mislabel a custom canvas as HSCREEN 2.

> Pause this owned run. Capture the normal framebuffer, then use capture_playtest_graphics for the declared back-buffer ID. Report which image is the last published frame and which is the paused register preview. Compare the canvas, viewport offset and physical identities without changing video registers or supplying an unapproved address.

A paused preview shows the registers and memory at one frozen state. It does not reconstruct an earlier scanout or guess an undeclared page. The project declares canvas origins, bounds and layout in verified build evidence; this capture tool accepts a buffer ID, not arbitrary memory addresses.

Keep a save between test sessions

After an app update or a later connection, `list_playtest_disks` finds the project's recorded test disks. The assistant can select one by its retained ID and exact hash when starting a new test. That cold-boots a separate writable copy; use the game's normal Load or Continue command to restore its saved data. The retained original stays unchanged; stopping the resumed run creates its own retained ID with a parent link. A recorded disk requires the same base artifact bytes. `artifact_disk_sha256` identifies the current produced disk, `disk_original_sha256` identifies the saved copy initially loaded into this run, and `disk_sha256` tracks its current writable contents. These need not be equal after the game saves. Older retained disks require their previous run ID and exact hash and report `legacy_receipt` origin: the earlier artifact provenance was not recorded and is not invented by rebuilding today.

To examine files in a saved disk yourself, keep a separate copy and use **Media → Floppy Disks… → Open Disk Image…** for the native Disk Image Editor. You can browse supported DECB or OS-9 directories and export the game's saved files. Keep the app-managed retained disk unchanged: its exact hash belongs to the resume receipt. Editing a separate copy does not turn it into a new retained MCP selection. Applying edits to a mounted disk is a separate local action that locks the machine and requires a confirmed restart.

Choose a stop the build can prove

For a project with verified debug evidence, the assistant can start at a named C function, pause or instruction-step its owned run, inspect locally configured named watches, and read only the advertised stopped-program memory ranges.

The build receipt's `source_debugging.functions` lists names with a mapped C statement. `source_debugging.function_entries` lists verified executable symbols, including assembly routines. Choose a name from the matching list. An assembly input sampler can use `stop_kind: "function_entry"`: the snapshot reports its real symbol, current address, RAM bank, registers and declaration location. It leaves `source_stop` empty where no C statement is mapped, and instruction steps there are labeled `assembly_instruction`. A declaration line is not the executing C line. Verified shared globals remain available; frame locals are not guessed.

An optimized Release build may have verified executable symbols but no typed C frame metadata. Its receipt reports `source_only`, with `executable_symbols` and unavailable variables. A verified symbol can still stop execution and expose registers and permitted code bytes. Such snapshots report `symbol_stop` and `executable_stop`; they do not invent a C function range, watch value or frame. Use the selected build's own evidence, never addresses borrowed from Debug.

A fresh C playtest connection can also advertise `get_call_context`. After `get_debug_snapshot`, use that same run and snapshot ID to read its bounded 64-byte window at the guest's current stack pointer. Raw words remain uninterpreted. Caller and integer argument values appear only where the compiled metadata and call instructions prove them; optimized or unknown layouts stay unavailable. A step, restart or rebuild requires a new snapshot. This reads only the owned disposable guest and never edits its registers or memory.

Read the error that stopped the build

When a build fails, read the build receipt's log as well as source diagnostics. It preserves bounded combined compiler, linker and Make output, the exit status and whether output was truncated. A warning list alone may omit the fatal reason. On a diagnostics-enabled connection, ask for the first actual error and any remaining build-log pages. The receipt distinguishes a compiler, assembler, linker or Make failure when the retained evidence identifies it. Successful builds can still carry warnings. Debugging receipts list verified C-line stops, function entries and available global declarations; unavailable values need an explanation rather than a guessed result.

> Read get_build for this failed build. Report first_error, failure_stage and exit_code. Use get_build_log and its next_offset to read the remaining retained output. Distinguish the actual error from warnings, and say whether any process output was already dropped. Propose a repair only after locating the cause.

The log combines compiler, assembler, linker and Make output; it does not pretend to separate stdout and stderr. A zero process exit can accompany a later `validation` failure when the produced artifact is unacceptable. If Make hides a command, the receipt cannot invent its text. Pagination retrieves retained output, not bytes discarded at the process-output limit. Keep the build ID and configuration with the diagnostic so the next repair targets the right result.

Hear the guest's next interval

An audio-enabled disposable connection can also capture a short WAV directly from the guest. Ask it to press the sound-producing key normally, capture the following interval, and report its field range, boot and sample count. Audio capture advances the guest and leaves it paused; a screenshot does not advance it. Retrying the same audio request returns the original clip without playing the interval again. This records no microphone or personal emulator session.

> Discover whether capture_playtest_audio is available. Use ordinary guest keys to trigger a sustained tone, then capture its following 60 fields. Report the WAV's sample rate, channels, sample count, boot and actual field interval. Tell me whether the clip contains changing samples. Keep the original request ID if delivery is interrupted; do not play the interval twice.

Each request accepts 1–120 fields. The result is a standard MCP WAV audio block: 44,100 Hz stereo, signed 16-bit PCM, before the Mac's volume control. Audio capture releases keys and drains preceding audio before advancing, so a short beep that ended during the input plan may already be over. A clip can end early at a verified stop. Check `fields_advanced` and `completed`; do not call an unheard or clipped interval a successful sound test. A constant DAC level can have a DC offset without audible variation.

Each run has bounded field, restart, mutation and audio allowances. Read `remaining_mutation_requests`, `remaining_audio_captures` and `remaining_audio_bytes`, as well as field and boot allowance. Audio is limited to 16 clips and 4 MiB across the run's boots; a new clip needs room for a maximum clip. `stop_playtest` remains available for cleanup and disk retention even after execution requests are exhausted. If retention is pending, keep the app open and retry the original stop request until the receipt confirms the disk was saved.

Keep observation separate from a debugger edit

> Read this build's two function lists. Stop at the input sampler using a verified function entry, report the symbol and physical bank, and inspect my configured player-state watch. Execute one instruction and report what changed. Do not alter registers, inventory or position through the debugger.

Those controls do not write registers, move the character or alter inventory through the debugger. Report build, launch, source debugging and gameplay results separately; a compiled program alone proves none of the later steps.

Rename a function across files

Reopen and share your repaired Hello Disk practice copy. Ask ChatGPT to rename `showHeading` to `showDiskHeading` throughout the project in one proposal. Look for a declaration in `include/hello.h`, a definition in `src/display.c` and a call in `main.c`. Open each file in the review window's **File** menu. Check the return type, arguments and spelling at all three places. **Apply All Changes** updates the drafts together. **Cancel Changes** keeps the current drafts. If you add a new project file yourself, use the connection's project refresh and approve the added file locally; older connections need a fresh export. Rebuild the applied revision and run it to check that your rename preserved the behavior.

Move a block one cell to the right

Chapter 5 of 9

On a 32-column CoCo text screen, the upper-left cell has a memory address, `$0400`. The next cell is at `$0401`. In this lesson you'll ask ChatGPT to move a block between them. You can check its work one instruction at a time and tell exactly when the CoCo display changes.

Put a byte on the screen

Open **Develop → Assembly Development...**. For the exercise, use a spare Disk BASIC machine at its `OK` prompt in 32-column mode. If you need to restore that width, type `WIDTH 32` at the guest prompt and press Return. Save any program you want to keep before loading another.

Make a writable copy of the screen-cell source. Choose **File → Open Source...** in Assembly Development and open the copy. Set the assembler target to **MC6809**. This is the processor the program is written for. It runs on either an emulated MC6809 or an emulated HD6309, including when your Mac itself has an Apple M-series chip.

```
        org     $3f00
start   lda     #$8f
        sta     $0400
        rts
        end     start
```

A dollar sign introduces a hexadecimal number. The first line tells LWASM, the assembler, to place the machine instructions starting at address `$3F00`. The word `start` names the first instruction. The last line tells the assembler where execution should begin.

Read the three instructions in the middle as a set of directions. `LDA #$8F` puts the byte `$8F` into A, a small storage location inside the CPU called an *accumulator*. The `#` tells the CPU to take that value directly from the instruction. `STA $0400` copies A to the screen address. The display hardware interprets that byte as a solid block. `RTS` returns to the caller. The VCC Plus loader sets up a BASIC caller for this program.

You can find the original exercise under **Quick Reference → First Program — Change a Screen Cell**. That example branches to itself so you can inspect the result for as long as you like. The book's version ends with `RTS` so you can also practice returning to BASIC.

Follow the first two instructions

1. Click **Assemble**. Open **Build details** and find `start` in the listing at `$3F00`. Assembling creates the binary. The CoCo starts executing it when you launch it.
2. Choose **Debug from Entry** and confirm the load. The program counter, labeled **PC**, should read `$3F00`. It points at the instruction the CPU will execute next.
3. Click **Step** once. A contains `$8F` and PC reads `$3F02`. At this point the program has only loaded A. Before taking the second step, locate the upper-left cell on the guest display.
4. Click **Step** again. The store has executed and PC reads `$3F05`, the address of `RTS`. Look at the upper-left cell. In the debugger's mapped memory view, the byte at `$0400` is now `8F`. If the cell already showed a block, the byte still lets you verify the store.
5. Choose **Continue**. The CoCo returns to `OK`. BASIC can now write to the display again, which is why the time to inspect that byte was just before `RTS`.

Give ChatGPT one instruction to change

In the assembly editor choose **File → Local Assistant Access...**, then **Save Assembly Connection...**. Open the connection with **Add to ChatGPT** and start a Work chat for this project. Give ChatGPT a job with a visible result.

> Read my shared assembly source. Tell me which instruction chooses the screen cell. Propose a one-line edit through VCC Plus that moves the block one cell to the right. Keep the value and the MC6809 target the same.

In the review, look for `sta $0400` changing to `sta $0401`. The other two instructions still have work to do. If the proposal instead changes the number after `#`, it changes what gets displayed. Cancel that edit and ask for the destination address to be changed. Choose **Apply Edit** when the proposed instruction is right. Save the practice copy.
Ask ChatGPT to assemble the **current shared revision** and report its entry point and any diagnostics. In the listing, place the caret on the executable `rts` line and choose **Set Stop**. Bring the disposable machine back to the 32-column BASIC prompt. You can type `CLS` there first to make the new block easier to see. Choose **Run Shared Build...** in the access panel and confirm the revised build. Each run starts with your local approval. A successful run of the earlier binary cannot verify this edit.

> At the stop for the run I just started, read the source line, PC and register A. Read the three instruction bytes at $3F02 using the current debugger snapshot. What screen address does that instruction specify?

The stopped instruction should be `rts`, PC should be `$3F05`, and A should be `$8F`. The three bytes are `B7 04 01`. The assembler encoded the destination address in the last two bytes. Build 30 shares the mapped instructions and the first screen byte with the assistant. You can inspect the second screen byte yourself in the native debugger at `$0401`. It should read `8F`. When the quiz in the next chapter puts three full lines of answers on screen, the CPU will still be doing work a byte at a time.
Choose **Continue** to return to BASIC. If the debugger reports that a snapshot is stale after you continue or edit, start the current build again and take a new snapshot at the stop.

Try the second processor

1. Save the revised assembly source and anything you want to keep from guest RAM. On the spare machine, open **Machine** → **Machine Settings...** → **Hardware** and choose **Hitachi 6309**. Choose **Apply and Restart...** and confirm. The processor change cold-restarts the CoCo. Wait for Disk BASIC's `OK` prompt.
2. Back in Assembly Development, change **Target** to **HD6309**. Ask ChatGPT to check the actual guest processor with `get_machine_status`, then assemble the current revision and report the assembler target. Check **HD6309** in both places. If the project needs reconnecting, save its connection again before requesting the build.
3. Set the stop on `rts` for this build and choose **Run Shared Build...** again. The earlier approval and snapshot belonged to the previous run. Inspect the new stop. Do you still see PC `$3F05`, A `$8F`, instruction bytes `B7 04 01` at `$3F02`, and `8F` at `$0401` in native memory? Those are the expected results. Both processors implement these instructions with the same encodings.
4. Choose **Continue** and check for the BASIC prompt. Revoke the assembly connection when you finish. To resume a practice setup that expects an MC6809, save any guest work and restore the processor in Machine Settings and the target in the editor.

Build a quiz and give ChatGPT the keyboard

Chapter 6 of 9

"What is the capital of Kentucky?" makes a good test question. A quiz that offers Louisville as a choice should let you pick it, tell you the capital is Frankfort, and count the miss exactly once. You can check all three behaviors by playing. You'll use the same test for a BASIC listing, a C program and a 6809 assembly program.

Play a round from the floppy

Choose the capitals game disk and save a copy as `CAPITALS.DSK`. Start a spare **Disk BASIC** machine and mount the image in drive 0. You can write-protect it. Give the CoCo display keyboard focus. At the `OK` prompt, type one of these commands and press Return.

Game	Command at the CoCo prompt	After the fifth question
Extended Color BASIC	`RUN"CAPITALS"`	Press **1** for another round or **2** to quit.
CMOC C	`LOADM"CAPSC":EXEC`	Press a key for another round. To switch games, restart the spare guest and return to BASIC.
MC6809 assembly	`LOADM"CAPSASM":EXEC`	Press **Y** to replay or **N** to quit.

Press Space at the title and wait for question one. Try **0** while the quiz is showing its three choices. You should still be on the same question. Choose **1**, **2** or **3**, read the feedback, then press Space to advance. There is no need to press Return after each choice. These games watch for individual keys. Pressing Return again can carry you past the feedback you wanted to read.

The disk is a convenient way for *you* to play. To let ChatGPT take a turn, open one of the editable projects below and start its shared run. You can do the rest of the chapter without mounting the floppy.

Give the writer a playable brief

The three versions began in separate ChatGPT Work chats. Each chat had access to one VCC Plus workspace. If you want to write your own version, start with one language. Save any existing work and use a new practice draft or project folder. Here's a brief you can reuse.

> Make a US state-capitals quiz for a CoCo 3 with a 32-column text screen. Include all 50 states. Ask about five different states per round. Offer three distinct capitals numbered 1, 2 and 3 and vary the position of the right answer. Retry an invalid key on the same question. After a miss, show the right capital. Wait for a key before clearing feedback. Show a score out of five and offer another round. Submit the complete source through my VCC Plus connection for review.

Add one of these directions, depending on what you've opened.

- **BASIC.** Ask for a numbered Extended Color BASIC listing using `INKEY$` for single keys. Ask the writer to reserve enough string space for the names and keep variable names distinct in their first two characters. In the saved BASIC listing, line 20 reserves 5,000 bytes and line 40 reads the 50 pairs into arrays. A typo in the pair for Kentucky would affect every round that draws Kentucky.
- **C.** Create a project with an editable `main.c` in C Development. Ask for a CMOC program using the bundled CoCo library for keyboard and screen I/O. Ask ChatGPT to propose the complete replacement for `main.c` through the project connection. The saved C source keeps a state's name alongside its capital in a

`Place` structure. Download the complete CMOC project for the same source and the Makefile VCC Plus uses to build it.

- **Assembly.** Start an assembly project with a `main.asm` and select **MC6809**. Ask for LWASM source with an entry address, the Disk BASIC ROM's character-input and output routines, and an `RTS` path for quitting. The saved assembly source uses `POLCAT` to poll the keyboard and `CHROUT` to print a character. It uses the time you spend on the title screen to choose a starting state.

At review, read the Kentucky entry and follow one path from a wrong choice to the score. In particular, look for an opportunity to count the same question twice when a player tries an invalid key. Apply a proposal you want to try, then save the draft or project. A chat containing a pasted listing can be useful for discussion. The editable game you can reopen lives in the saved project.

Run the saved sources with ChatGPT

Start a fresh Disk BASIC session in VCC Plus. Use a disposable machine with a floppy controller, the cassette ejected, and mounted disks ejected or write-protected. If C reports that a BASIC program is already in memory, save any listing you want to keep and start with an empty BASIC session. Don't clear an unfamiliar program to get past the warning.

1. **BASIC.** Select CAPITALS.BAS in the Mac reader to open an editable copy in BASIC Development. In a browser, save the listing and open it in **Develop → BASIC Development...**. From that editor's **Local Assistant Access...**, enable **Allow proposed edits and controls for runs I start** and save a review-and-run connection. Use **Add to ChatGPT** and start a Work chat. Opening an existing listing is enough for manual play. For the shared run, VCC Plus also needs a proposal you've applied to the current revision. Ask ChatGPT to change only the comment on line 10 to `10 REM US STATE CAPITALS QUIZ`. Compare the two lines, choose **Apply Edit**, then **Run** on the applied proposal in the access panel. To play the exact original over MCP instead, begin with an empty practice draft and have ChatGPT submit the supplied numbered listing verbatim. Compare it with the saved file before applying and running it.
2. **C.** Save the CMOC project and expand the ZIP in Finder. Open the `CapitalsCMOC` folder containing `main.c` and `Makefile` in **Develop → C Development...**. Choose **Assistant...**, save its private project connection and use **Add to ChatGPT** in a fresh Work chat. Ask ChatGPT to build the current shared project and report any diagnostics. After a successful build, choose **Run C Build... → Play in CoCo**. The other choice, **Inspect at main**, is useful when you want to stop at the first source statement. For this round, wait for the title and the prompt to start.
3. **Assembly.** Save main.asm, then open the saved file in **Develop → Assembly Development...**. Confirm **MC6809**, then use **File → Local Assistant Access... → Save Assembly Connection...** and add this connection to its own Work chat. Ask ChatGPT to assemble the shared revision. Check the successful build and entry point `$3F00`. Clear any source stop you set while experimenting. With the guest at the BASIC prompt, choose **Run Shared Build...** and confirm the launch.

Each choice above is a complete route to a game. You can test one language today and come back to another later. If you do test all three, give each connection a distinct file and return the disposable guest to a clean BASIC prompt between games.

Watch someone else take the quiz

Leave the quiz on its title screen. In the Work chat connected to that game, give ChatGPT both the controls and a stopping point.

> Play one round using the active VCC Plus run. Read the guest screen and send the game keys yourself. Start with Space. Try 0 on the first question, then answer it correctly. Deliberately miss question two and read the capital the game gives you. Answer the remaining questions correctly. After each answer, read

> the feedback before sending Space. If the screen is slow to change, read it again. Stop at the replay prompt. Report the missed state, its capital and the score exactly as displayed.

The player obtains the active run ID with `get_run`, reads with `read_screen_text`, and sends a single letter, digit or space with `send_keys`. Each intended keypress gets its own request ID. A retry of the *same* key request reuses that ID so it won't count as another press. Give the player time to observe the screen between presses. If it stalls, ask what the guest **currently displays**. Seeing the replay menu yourself is a good way to check the final report.

What happened on the M2

On November 9, 2026, the first complete submissions for all three languages ran in VCC Plus build 29 on the M2 Mac. The operator reviewed the sources, authorized the runs and played the games. The original BASIC listing reached its title screen. Bundled CMOC and LWASM each built its first submission with zero diagnostics. The operator played each round to **4/5**. All three saved game sources are byte-for-byte copies of the first proposals. Independent emulator runs on both guest processors also brought every one of the 50 states to the screen in each language.

With VCC Plus 0.1.0 build 30, the operator set up three fresh connections on the M2, reviewed and started BASIC, and built and launched C and assembly. This time the stock **ChatGPT for Mac** app sent every game key in three separate Work chats. A separate observer read the guest screen. Each player tried **0**, deliberately missed the second question, and stopped at the replay prompt.

Saved game	Correction displayed after the miss	Score in the guest
BASIC	Rhode Island — Providence	`YOUR SCORE: 4 /5`
C	North Carolina — Raleigh	`FINAL SCORE: 4/5`
Assembly	Kentucky — Frankfort	`SCORE: 4/5`

```
YOUR CHOICE (1-3)?
CORRECT!
PRESS ANY KEY TO CONTINUE

QUESTION 5 OF 5
STATE: PENNSYLVANIA
1. SACRAMENTO
2. HARRISBURG
3. BOSTON
YOUR CHOICE (1-3)?
CORRECT!
PRESS ANY KEY TO CONTINUE

SCORE: 4/5
PLAY AGAIN? Y OR N
```

Figure 4: Assembly quiz showing the correct answer to the fifth question, Pennsylvania, followed by SCORE 4/5 and PLAY AGAIN? Y OR N.

On that last screen, the assembly game is still running and waiting for a decision. If you want to test its exit path, ask ChatGPT to send **N** once. The program prints `THANKS FOR PLAYING!` and returns to BASIC. A finished run can report its final screen. Its game-key controls close when it returns. Revoke the connection after your round.

Grow the idea into Capital Quest

The five-question games above make the connection easy to understand. A later project, **Capital Quest**, took the same subject much further: every state in one 50-question quiz, a color map, typed answers and a review of mistakes. Its public CoCoCapitalQuiz repository contains the editable C project, map inputs and credits, a ready-to-play `CAPITAL.DSK`, sample screens, tests and the development brief. This is a separate game from the book's `CAPITALS.DSK` and its three short examples.

The core prompt generated an almost feature-complete program with ChatGPT Codex using GPT-6.1 Sol High. The initial delivery already included both quiz modes, all 50 states, map highlighting, spelling correction and a review of missed answers. VCC Plus supplied the installed toolchain and public MCP interface for building and playing it. The target is a **512 KB CoCo 3 with an MC6809**, with a 320 × 192, 16-color map and Northeast/Hawaii detail views.

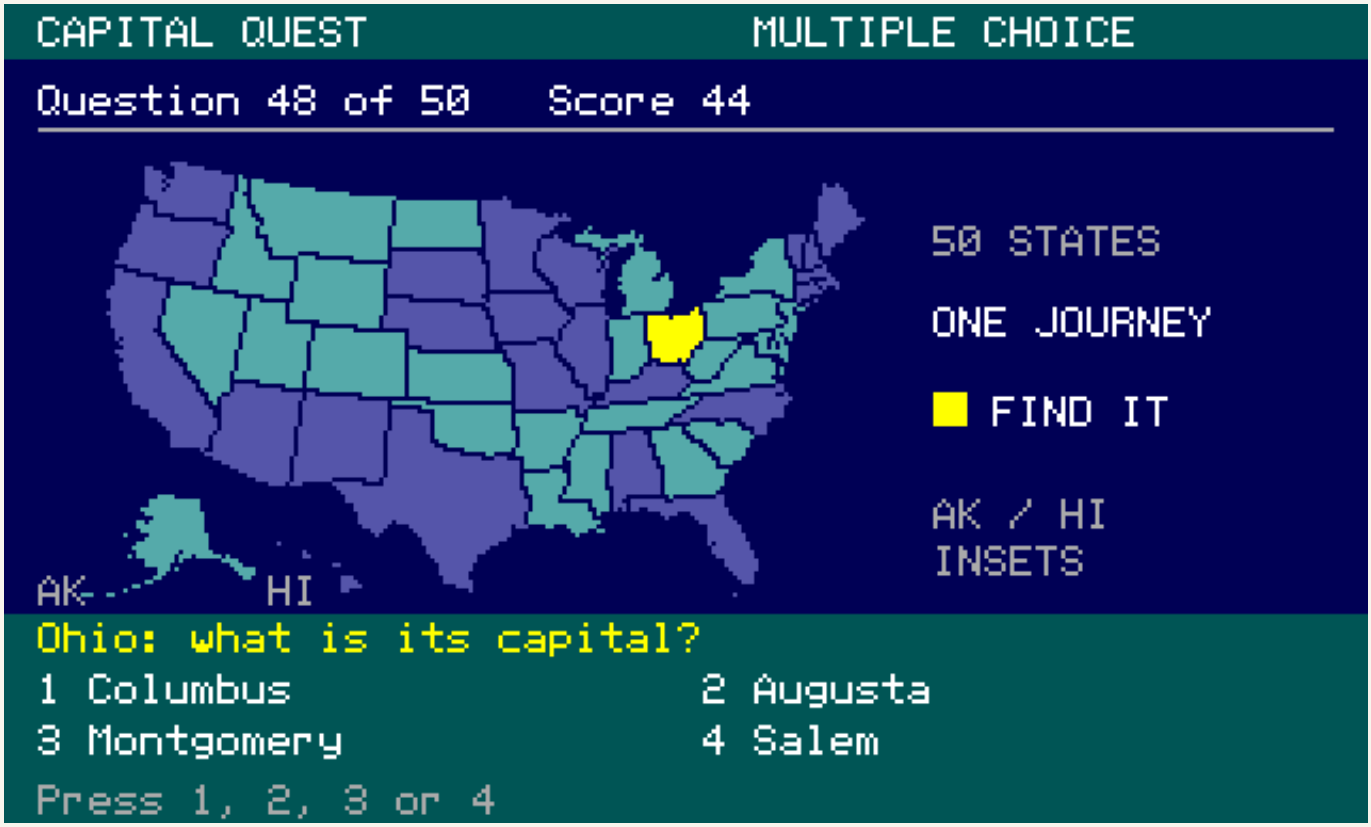


Figure 5: Earlier Capital Quest edition: Question 48 of 50, score 44, Ohio highlighted on the national map, and Columbus among four capital choices.

The map is adapted from © Fla-shop.com, under Creative Commons Attribution 4.0. The adaptation reduces resolution, simplifies boundaries, recolors and encodes the drawing data, and adds detail crops. The repository's `assets/CREDITS.md` retains the source, license and adaptation notices; **C** on the game's title opens its credits.

Give a larger project a richer brief

Open the project's complete CHATGPT_PROMPT.md to see what the writer was asked to deliver. Notice how much more it settles than "make a capitals quiz." It specifies behavior, constraints, evidence and what to preserve while working.

Part of the brief	What it lets you check
Ask all 50 states exactly once, numbered 1–50	A complete round can reveal duplicates, omissions and off-by-one numbering. Alaska and Hawaii are included.
Offer four distinct choices and a typed mode	Both the menu player and the line editor need real tests.
Accept case and spacing variants, but handle a near spelling carefully	A close answer gets a correction opportunity, without an early point or an immediate answer reveal. Another state's exact capital remains a wrong answer.
Highlight the question's state and enlarge small regions	The map must agree with the data and remain useful at CoCo resolution.
Preserve attribution and convert graphics before running	The guest uses compact drawing data; it does not try to render an SVG.

Part of the brief	What it lets you check
Use ordinary guest input and actual framebuffer captures	The test must observe the running program, rather than setting its score through a debugger.
Keep source, assets, build instructions and test limitations	Another reader can rebuild, inspect and improve the result.

The brief also asks the assistant to discover the installed tools and consult the manuals before choosing APIs. It gives the writer room to make routine design decisions while keeping the target machine, program behavior and testing method clear. You can use the same pattern for a word game, a music lesson or a small business application: describe a complete experience, then give it observations that can establish whether that experience works.

If you reuse the brief to create your own version, change its checkout path and read its assumptions first. It describes the original empty project. Today's download already contains a game. Use a new practice folder when you want to start over; preserve the supplied project when you want to study or extend it. The current prompt also includes a later **Addendum** for sound and automatic multiple-choice progression.

Follow the build and test loop

Open the project in C Development and use its Make recipe to combine C sources with the prepared map data. In **Assistant...**, enable **Use Project Makefile** and **Allow Disposable Playtests**, review the shared inputs and save a private connection outside Git. Begin with live tool discovery and `get_capabilities` so the assistant identifies the project, revision and available build choices.

Ask it to build the declared disk action in a named configuration and read `get_build`. When the build succeeds, use its artifact ID with `start_playtest` and poll `get_run` until ready. This creates a separate writable test disk and 512 KB guest, preserving the personal machine. Send ordinary physical keys with `playtest_input`, include release fields between presses, and inspect the actual framebuffer with `capture_playtest`.

Capital Quest's tests compare drawn text against the font in the matching build and selected-state pixels against the map masks. You can also inspect the pictures yourself: does the named state light up, are the four choices readable, and does the displayed score change when it should? For a logic question, use the build's advertised C-line or function-entry evidence and read-only snapshots. For sound, `capture_playtest_audio` supplies the guest's following bounded interval as a WAV. Chapter 4 explains its timing and key-release behavior.

Host logic tests and guest play complement each other. Host tests exercise thousands of shuffles and wrong-capital pairs quickly. Full guest rounds check input, screen layout, numbering, feedback, scores, review, replay and exit. The project's `Test` configuration uses seed **2709** for repeatable rounds; ordinary `Release` play mixes timing and start input into its seed.

Keep each result with its build ID, configuration and artifact. Choose Test when reproducing a particular sequence, and finish its run before building another configuration. Save evidence in a declared output folder or outside the watched checkout so captures remain separate from program inputs. Use scoped project refresh after changing shared files or build controls, as explained in Chapter 4. Stop the owned worker when you finish and keep its retained disk receipt with your observations.

See what the complete program delivers

A rich brief gave the project its shape. The core program shuffles all 50 states, offers four-choice or typed play, keeps the highlighted map associated with each question, and finishes with a score and paginated review. A close spelling lets the player edit or reveal; a successful correction earns one point and is tracked separately. This is enough to make a complete learning game from the original idea.

The next refinement added rising tones for correct answers and falling tones for misses using the CoCo's built-in sound. Multiple-choice feedback advances automatically after a readable pause, while typed mode keeps

Space-to-continue. The prompt's Addendum captures that request. It is a useful example of shaping an already working program through a focused follow-up.

Further refinements made the map show the player's answer history: light teal for unanswered states, purple for correct answers, red for misses, and yellow for the current question. The current state stays yellow through feedback and takes its answer color when the next question opens. Northeast and Hawaii details follow the same rule, and a new quiz clears the history. These colors have no regional meaning.

Another follow-up changed multiple-choice placement. The first correct answer may occupy any slot; each next question chooses randomly from the other three. It excludes the previous question's correct slot even when the player selected a wrong answer. Host tests checked 100,000 question sets, and complete guest rounds checked the choices actually displayed. This is a useful refinement to test separately from shuffling the state order.

The project's test records include these complete rounds:

Playtest	Recorded outcome
Multiple choice with deliberate misses	All 50 states once, numbering 1–50 and a final score of 42/50.
Typed answers with spelling exercises	All 50 states once, 47/50 and two successfully corrected spellings.
Ordinary Release play	A complete 50/50 round and the corresponding review.

The mixed-answer rounds exercise scoring and feedback as well as correct answers. Tests also cover casing, multiword capitals, aliases, Backspace, long input, replay, held-key handling and clean return to BASIC. Read `docs/TESTING.md` and `docs/FEEDBACK_TESTING.md` for the original scenarios, and `docs/MAP_PROGRESS.md` and `docs/ANSWER_SLOTS.md` for the later map and slot checks. Those are dated development records. See `docs/evidence/` for the retained receipts.

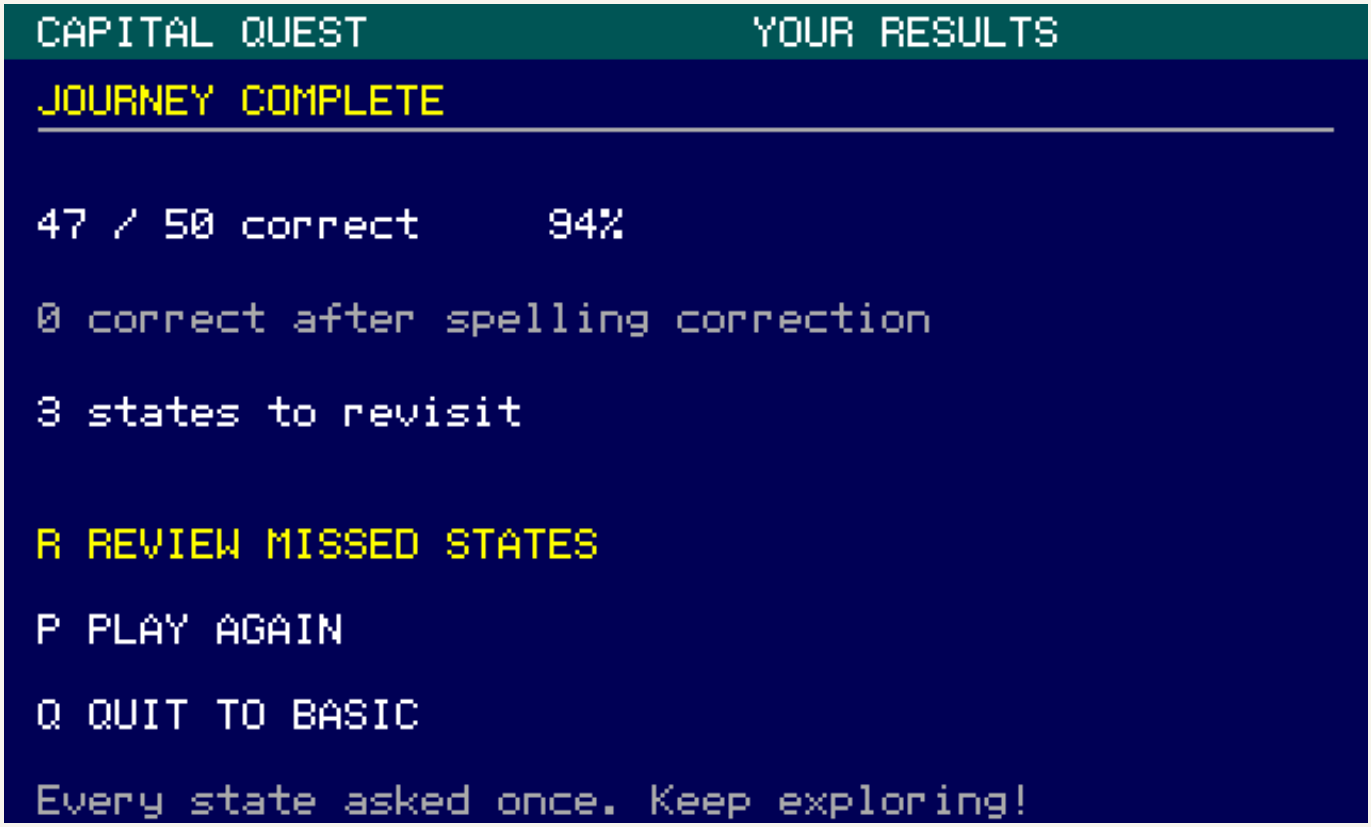


Figure 6: Earlier Capital Quest results: 47 of 50 correct, 94 percent, zero spelling corrections and three states to revisit, with review, replay and quit controls.

This results screen gives the player a useful next action: review the three missed states, play again or return to BASIC. The same idea belongs in your own program's completion screen. A score is more useful when the player can see what to practise next.

Try the delivered game, then change one rule

1. Read the project's README and player manual. To play the delivered disk, mount your own `CAPITAL.DSK` in **FD0** of a **CoCo 3 with 512 KB RAM and Disk BASIC**. Preserve existing work, choose **Cold Restart**, wait for BASIC's `OK` prompt, then type `DOS`, or `LOADM"CAPITAL.BIN":EXEC`. RGB is the tested color presentation; both emulated MC6809 and HD6309 have been checked. Keep personal media separate. This route needs no assistant connection.
2. To inspect or rebuild, obtain the repository files through your browser or your usual Git client. Open that folder in **Develop → C Development...**. Select **Release**, then **Build**. The declared target builds the disk; use **Show Build Products** to locate this build's `CAPITAL.dsk` and binary. Preserve the map's `assets/CREDITS.md` and source notices with your copy.
3. To repeat the assistant workflow, use **Assistant...**, review the shared inputs, select the Make and disposable-playtest options, and save a fresh private connection. Discover its tools and ask it to identify the project, current revision and advertised configurations before building.
4. Ask for a short observable check before a full round. The prompt below connects the current tools to a behavior from the original brief.

> Build this project's advertised Test disk and start a disposable 512 KB MC6809 run of that artifact. Poll startup until ready. In typed mode, read the state from the actual framebuffer, try a close spelling, and capture the correction prompt before fixing it with ordinary keys. Show that the score changes once. Report what passed, what you could not observe, and the retained disk receipt when you stop. Do not change guest memory or registers.

The **November 12, 2026 game edition** uses **1** for multiple choice, **2** for typed answers and **C** for credits. **Q** (or lowercase **q**), **Esc** and CoCo **Break** quit to BASIC from every screen, including questions, spelling help, feedback, review and the automatic reading pause. VCC Plus maps the Mac's **Esc** key to CoCo Break. Release each key before the next action.

Multiple-choice answers use **1–4**. Typed answers use letters, spaces, periods, Backspace and Enter. Press **Enter with an empty answer** to reveal the capital and count the question as a miss; clear any text with Backspace first. A near spelling offers **R** to edit or **V** to reveal. Q, Esc and Break quit rather than revealing an answer. Typed feedback waits for **Space**; multiple-choice feedback advances automatically. After a complete round, **R** opens the missed-answer review, **N** advances its page, **B** returns to results and **P** starts again.

Use the matching disk and player manual together. An existing Getting Started installation or personal source copy can contain an earlier edition. Reading the new online manual does not replace that disk. To use these quit controls, download the November 12 edition through the project's player guide and load a separate copy; preserve your existing disks and edits.

Now choose one small refinement of your own. Perhaps you want a longer reading pause or a different highlight color. Predict which observation should change and which should remain the same. Review the source edit, build the new revision and repeat that check on its actual artifact. The useful outcome is a program you can inspect and improve, together with evidence that explains its behavior.

Find the first wrong result

Chapter 7 of 9

Suppose a game awards one point for a correct answer. You start with seven points, answer once, and finish with nine. Which part needs fixing? The score might have gone up twice, or the game might have printed the wrong number. You can settle that question before changing a line.

Stop before the score is printed

Use a separate BASIC draft for this exercise. Save any work you want to keep in the editor and on the guest. Share the practice draft with review-and-run access. Ask ChatGPT to propose the following listing exactly, including the deliberate scoring mistake.

```
10 S=7
20 S=S+2
30 PRINT "SCORE";S
40 END
```

1. Review the four lines in VCC Plus and choose **Apply Edit**. In the BASIC editor, click the gutter beside line 30 to set a breakpoint. Open **Variables and Watches** and add the watch `S`.
2. Start the reviewed program with **Run** in its assistant panel. When line 30 is highlighted, lines 10 and 20 have executed. The `PRINT` at line 30 is still ahead of you.
3. Read `S` in the native workspace. Ask ChatGPT to make its own observation.

> Use the current BASIC run and get_debug_snapshot. Which numbered line has the guest reached? What value does my watch S show? Wait for me before proposing a change.

The expected stop is line 30 with `S = 9`. The guest already holds the wrong number before it draws the score. **Continue** in VCC Plus lets the pending statement print it. Wait for the guest to return to its BASIC prompt.

> The rule is one point per correct answer. Use the current revision to propose the smallest change that fixes the measured error. Keep the starting score and the display line.

The repair you want to see is `20 S=S+1`. Check the line in the native review. Apply it, confirm the breakpoint and watch are still set, then start a **new** reviewed run. The watch should now show eight at line 30. Continue and compare that value with the score printed in the guest. Save the corrected draft if you want to keep it. You have tested both places where the error could have appeared.

Choose a repair the evidence supports

A compiler diagnostic can identify a name that the compiler cannot resolve. It cannot tell you that a game should award one point. In the C version of this exercise, an expression such as `score += 2;` builds successfully. A request for a diagnostic-backed C line repair would have no diagnostic to attach.

For a scoring mistake in a shared C or assembly project, ask for the source statement responsible and a project edit. Even a project edit can touch just one file. Read the before and after text in VCC Plus. Does the change correct the arithmetic, or does it merely print the number you hoped to see?

> My rule is one point per correct answer. The watch shows the wrong score before printing. Read the current project and propose one change to the calculation. Tell me which input will distinguish that repair from printing a fixed number.

When you apply a project repair, ask for a build of its new revision. Start the resulting program locally and replay the same input. A second run with a different starting value will test whether the repair generalizes.

Read the stop you actually reached

For an ordinary C project, add a watch in the C workspace before starting the shared build. Choose **Run C Build… → Inspect at main** when you want to pause at the first verified statement in `main`. Choose **Play in CoCo** to run an interactive program. Play records the entry stop and then continues. A watch from that stop describes entry even if the game is now showing its fifth question.

Here is a recorded observation from a two-file C project in installed build 29. Build 30 retains these debugger tools. The project defined the unsigned integer `beaconWord`. A reader trying a different project will see different addresses and names.

Observation	What ChatGPT read through MCP
Verified source stop	`main.c`, line 8
Program counter (PC)	`$2E71`
Configured watch	`beaconWord = 19754`
Readable code in the snapshot	`$2E6D` through `$2E97`
Three bytes at the PC	`17 00 16` in hexadecimal

Try this with a watch that belongs to your project.

> Read the run I started. From get_debug_snapshot, report the source stop, PC and available watches. If three bytes starting at the PC fit inside one reported range, use this run and snapshot to read them. Say which values you observed.

`get_run` identifies the launch. `get_debug_snapshot` supplies the snapshot ID and the ranges captured at the stop. `read_debug_memory` uses both IDs. The dollar sign in `$2E71` marks a hexadecimal address. The three bytes in this example came from that address in the captured program. A watch marked unavailable is worth reporting as such. Select a mapped executable statement and an in-scope variable for your next native debugging session.

For assembly, a useful place to stop is the final `RTS` after a calculation and a store. Set the source breakpoint on the return instruction before choosing **Run Shared Build…**. At that stop the store has executed and the register used by the calculation still holds its value. In a tested MC6809 routine, A and the byte at `$0400` both contained `$29` (41). Check the readable ranges before asking for that byte in another program.

Compare the two processors

For a C project, **Allow Disposable Playtests** in its Assistant panel grants isolated runs of its built disk. A fresh connection can start a 512 KB MC6809 (the default) or HD6309. Ask the assistant to discover its current tools first. Changing the Mac's processor architecture does not change the emulated CoCo CPU.

> Build the current compatible project once. Start one disposable MC6809 run, stop at a verified function, and record the artifact, registers, available globals and framebuffer. Stop and retain its disk. Start a separate HD6309 run of that same artifact and repeat the observation. Report actual processor, execution mode, MD and any unavailable evidence. Use ordinary guest input.

The 6309 resets into emulation mode. Only a guest instruction changes it to native mode. Its debugger also exposes E, F, W, V and MD; a 6809 has no such registers. Selecting a 6309 does not rebuild your code or supply missing C metadata. A cold restart keeps the selected CPU and writable test disk; choose a new run to change processors. An early startup reply names the requested CPU; wait for a ready receipt before claiming the observed mode.

Both input plans and audio capture release physical keys when they finish. Audio capture also releases keys before recording its following bounded interval. A sound clip taken after a movement plan does not prove that the movement keys remained held throughout that clip.

Repeat one observation after a change

If you edit the draft while ChatGPT is preparing a repair, stop there. Have it read the current revision, compare it with the intended change, and submit a fresh proposal if the repair still belongs there. A new build can move a statement to another address. A reset, a different mounted medium or a new launch can invalidate an earlier run. If the app reports a stale run or snapshot, obtain the identifiers for the matching launch and stop again before reading memory.

For a last pass, choose one behavior the repair should preserve. With the four-line BASIC exercise, try a starting value of zero in your practice copy. Predict the score, rerun, and inspect the same watch before the same print. A result of one shows that the program adds a point to its starting score. Simply replacing the output with a literal eight would have failed this second check.

Recorded check. On both the MC6809 and HD6309 guests in the build 30 emulator core, line 30 stopped with `S` equal to 9, 8 and 1 in the three versions. After Continue, each guest printed that same score and returned to `OK`.

Keep a field guide at hand

Chapter 8 of 9

Use this page when a conversation and the CoCo seem to disagree. Start with a question the machine can answer now.

> Identify the VCC Plus draft or project this connection shares. Report the guest CPU and the first reason a local run might be blocked.

Find the next action

What you see	What to try
ChatGPT describes a different listing.	Ask for the shared title and first numbered line through `get_capabilities` and `read_source`. Share the intended BASIC editor through its own **Local Assistant Access...**. C and assembly each use a separate project connection.
The saved connection appears in settings, but a new chat cannot read the source.	Keep VCC Plus and the shared workspace open. Use **Check Setup** in the connection panel, open a fresh **Work** chat, and check `/mcp`. If you replaced the grant file, restart the corresponding server so the helper reads the new grant.
ChatGPT can read BASIC but cannot propose an edit, or an older connection lacks newer tools.	Save a fresh connection from the current app with the permissions you intend to grant. BASIC review uses **Allow proposed edits and controls for runs I start**; C has separate Make and disposable-playtest choices. Restart the corresponding server and discover its actual tools.
A proposal seems to have gone missing.	Ask for `get_proposal` using the returned proposal ID. A pending change awaits local review. If the draft has changed since the proposal, read its new revision and ask for a new proposal.
The C project builds in the workbench but the assistant build fails.	Check whether this grant uses the restricted recipe or the approved **Use Project Makefile** mode. Read `get_build.first_error` and the retained `get_build_log`. Inspect approved inputs and refresh locally changed build controls before trying again.
Build succeeded, but there is no run ID.	Check the run blockers in `get_machine_status`. In VCC Plus start the matching reviewed listing or shared build. For an interactive C game, choose **Play in CoCo**.
A run is waiting for a key.	Read the guest prompt. For a shared single-key menu, send one permitted key, then read again. An owned disposable run can use bounded physical input or `playtest_debug` to continue a verified stop; identify which kind of run you have first.
The guest asks for a whole line, or displays only a picture.	An owned disposable playtest supports `playtest_input` with letters, `space`, `period`, `backspace` and `enter`, plus framebuffer captures. Ordinary shared `send_keys` retains its smaller key set; type a whole line locally through **Input**

What you see	What to try
	→ **Focus CoCo Display** when that is the connection you are using.
A short compiled program has already returned to BASIC.	Ask for `get_run` and its optional `final_screen_text`. This is the text captured when the program returned. Live key and screen access have ended. A graphics-only run may have no final text.
A key, run or memory read is refused after a change.	Stop sending input. Check the project revision, the guest and its media. For a new launch get the new run ID. For a new debugger stop get a new snapshot and read within one of its reported ranges.
Startup took longer than a transport request.	Keep the reserved run and original request IDs. Poll `get_run` for phase and readiness. A ready receipt is separate from reaching a breakpoint. Use `stop_playtest` to cancel; do not create a second start request to guess whether the first worked.
A C function has no mapped stop.	Read `get_build.source_debugging`. A name in `functions` supports a C-line stop; a verified entry in `function_entries` can use `function_entry` without claiming a C statement. A source-only build may expose an executable symbol with unavailable variables.
The run has reached a request or audio limit.	Read the remaining budgets. Stop and retain the owned disk with `stop_playtest`, which is exempt from execution request limits. If saving is pending, retry the original stop request; keep the app open until retention succeeds.

Keep the program you already have

An editor draft on the Mac and a listing in CoCo BASIC memory can contain two different programs. If the C workbench says BASIC is occupied, save the open editor draft first. At a supported BASIC prompt, use **Read from CoCo** in BASIC Development, then **Save As...** to preserve the guest listing in its own file. Read-back saves program text. The variables and the place where a program paused belong to that execution.

Only when you want to clear the guest listing and its variables, type `NEW` yourself at the CoCo's `OK` prompt. Ordinary C needs empty BASIC memory. The Hello Disk verification also needs an empty floppy drive 0 for the disk it creates. For a shared run, finish recordings, eject the cassette, and eject or write-protect writable floppy and hard disks. End an active Mac shell session, turn off DriveWire networking, and use a BASIC setup with only the floppy controller if the status report calls for it. Handle any save or export prompt before ejecting your media.

Applying a proposal updates an editor draft. Use the workspace's **Save** command when you want to keep that change on disk. A grouped project edit has a grouped Undo while the applied revision is current. If you have edited either file since, inspect both current drafts before deciding what to restore.

What to ask for

The table describes current tools. Each workspace and saved grant exposes its own subset; discover that subset before asking for an operation.

Task	Tools and useful limits
Identify the workspace and machine	`get_capabilities` identifies the shared draft or project. `get_machine_status` reports the emulated processor, devices, build status and run blockers.
Look up a CoCo detail	`search_help` returns matching installed manual sections. Give `read_topic` a topic ID returned by the search. If the result is marked truncated, open that section in **Help → Manual Library...**.
Read source	`read_source` reads a BASIC draft. `read_project_file` reads a named shared C or assembly file. Open editor drafts can include unsaved changes.
Propose a change	BASIC has `propose_edit` for one existing numbered line and `propose_listing` for a complete numbered ASCII listing up to 12 KiB. C and assembly have `propose_fix` for a line and `propose_project_edits` for complete replacements of one to eight existing shared files, up to 16 KiB in total. A C line repair needs a matching compiler diagnostic.
Check the review or build	`get_proposal` retrieves the native review decision. In C and assembly, `build_project` requests a build of a revision and `get_build` reads its status and diagnostics.
Refresh a C project's inputs	`read_project_inputs` pages the approved manifest, up to 256 rows per page. `refresh_project` requests adoption or local review; `get_project_refresh` polls that decision. Neither executes the recipe.
Read the build failure	`get_build` reports diagnostics, first error, status and stage when known. `get_build_log` reads retained combined output in pages up to 32,768 UTF-8 bytes; follow `next_offset` and report truncation.
Deliver declared build files	On a separately approved export connection, `get_build.exportable_outputs` lists exact output identities. `export_build_outputs` selects the original build/artifact/revision and approved destination; poll `get_build_export`, then independently verify delivered lengths and hashes. It does not rebuild or export guest saves.
Play a text game	`get_run` identifies the run you started. `read_screen_text` observes the guest text. `send_keys` accepts one ASCII letter, digit or space per ordinary key request, up to 64 distinct accepted requests per run. Give each intended press a fresh request ID. Reuse the same ID and identical key to retry an uncertain delivery.
Inspect a stop	`get_debug_snapshot` describes a verified stop, available watches and readable ranges. `read_debug_memory` reads 1–256 bytes wholly within one advertised snapshot range. Device I/O is outside those ranges.
Inspect an owned call	When advertised by a fresh C playtest grant, `get_call_context` uses the same run/snapshot IDs for a fixed 64-byte view at actual S. Treat raw words as uninterpreted; accept caller/argument labels only with verified evidence. It does not grant arbitrary stack addresses or shared-run access.

These tools require the C connection's **Allow Disposable Playtests** grant and the corresponding live capability. They operate on the assistant's separate test machine, rather than the personal CoCo used by the earlier lessons.

Task	Owned-playtest tools and useful limits
Start or check preparation	`start_playtest` binds the current project, revision and artifact. `get_run` polls its reserved run. Fresh processor-aware grants accept `MC6809` or `HD6309`; the default is MC6809.

Task	Owned-playtest tools and useful limits
Enter a line or hold a chord	`playtest_input` accepts up to 64 steps, eight distinct physical keys per chord, 1–600 fields per step and 3,600 fields per request. `period` is a physical key; `backspace` and `left` name the same physical key. Empty `keys` releases input.
Test a joystick	Fresh connections accept optional `joysticks` per input step: right/left port, integer X/Y 0–16383 (center 8191) and `fire1`/`fire2`. Omitted ports center and release. `input_trace` records delivery; inspect actual gameplay frames for the result.
Diagnose input	`get_playtest_input` reads owned input without advancing. Shared `get_machine_status.input` distinguishes host preview, last submission and actual core input, including connection/focus availability.
Capture a picture	`capture_playtest` returns the last published framebuffer without advancing. At a paused state, `capture_playtest_graphics` can compare the current register-selected display with a verified declared buffer ID, including a bounded GIME canvas.
Capture sound	`capture_playtest_audio` advances 1–120 fields and returns a PCM16 stereo WAV, then leaves the guest paused. It uses no microphone. Read actual timing and remaining audio allowance.
Record gameplay for a manual	On a video-enabled connection, `start_playtest_video` starts a bounded MP4 clip from an owned playtest. Ordinary input drives the recorded guest fields; waiting between calls adds no footage. `stop_playtest_video` completes the clip, and `read_playtest_video` returns bounded chunks for saving and verifying it. Choose CoCo audio per clip. Mac windows, microphone and host capture cues are excluded. Inspect completion/truncation, field counts, byte count and SHA-256 before including a clip in a manual.
Pause or inspect code	`playtest_debug` offers `pause`, `continue`, instruction `step`, and verified named `stop_at`. Match `stop_kind` to the build's C-line or function-entry evidence, then request a fresh debug snapshot.
Use Home and Data together	Fresh multi-drive grants expose `drive` 0–3 on `exchange_playtest_media`, defaulting to FD0. Pause with the shared controller idle; use listed private-copy IDs and current hashes, optional insert-time protection, and per-drive receipts. Never silently move one writable copy between drives.
Test Save and Load	`restart_playtest` cold-boots with the current mount map and writable copies. `stop_playtest` retains every copy; `list_playtest_disks` finds project-scoped records. Resume by retained ID and exact hash under compatible base-artifact bytes, then use the guest's Load command.

Give each intended mutation a fresh request ID. Reuse the same ID and identical arguments to recover an uncertain delivery; changing its meaning is a conflict. An owned run has 216,000 field attempts, 16 boots and 1,024 mutation requests. Breakpoints can shorten plans, so use actual returned clocks. Keys and joystick buttons are released and axes centered when an input plan finishes. These are upper bounds, not a promise that a game will reach its intended checkpoint. Stop/retention remains available for cleanup.

BASIC also accepts one fixed probe, `PRINT 7` followed by a newline, to verify that the interpreter returned to its prompt. Type your own direct-mode commands at the guest keyboard.

For a manual lookup, start with terms a CoCo book would use.

> Search the installed manuals for INKEY$. Read a matching section and explain what the function returns when nobody has pressed a key.

Close or renew a connection

Use **Revoke Access** in BASIC or C, or **Revoke Connection** in assembly, when you finish sharing. Revocation disables the grant and removes its file. Text already returned to a conversation can remain there. If you want to work together again, open the intended workspace, save a fresh connection, and restart that server in ChatGPT. A different saved path can be registered through **Add to ChatGPT**. Keep the private connection file on your Mac.

Words you will meet

Guest means the emulated CoCo. **Host** means the Mac that runs VCC Plus. A host with Apple silicon can still be running an MC6809 guest.

Grant is the private connection file that authorizes access to a selected workspace. **MCP** stands for Model Context Protocol, the interface ChatGPT uses to call the tools offered by that connection.

Revision identifies a particular version of the editor draft or shared project. A **build** translates a compiled project revision. A **run** identifies one launch on the guest.

A **breakpoint** pauses execution at a chosen location. A **watch** is a named value you want to inspect there. A **snapshot** records the verified stop and the memory ranges the assistant can read from it. On the next debugging pass, ask which run and which stop supplied the value you are comparing.

Start a project you can hand off

Chapter 9 of 9

An idea can fit in one sentence: make a falling-block game, catalogue a disk, or teach the capitals. A project needs a few more decisions. Which CoCo will run it? Which controls must work? What may the assistant change? Where should it look when it does not know an API?

Write those decisions down once. They give your first chat a useful starting point and let another chat continue without reconstructing the whole conversation. You can change them as you learn; keep the changes visible.

Copy the starter, then make it yours

Download the Project Starter folder into your new project. These are editable documents, not a finished program or an approved build recipe. Replace the bracketed fields, remove irrelevant sections, and review the rules before adopting them.

File	What belongs there
`PROJECT_BRIEF.md`	The program you want, target machine, controls, scope and acceptance checks.
`AGENTS.md`	Working rules that the owner has reviewed and adopted.
`START_HERE.md`	A short first request referring to those documents.
`MCP_SETUP.md`	Connection checks and limitations, without private grants or credentials.
`TESTING.md`	The exact program, machine and results actually tested.
`DECISIONS.md`	Measured limitations, approved exceptions and the reasons for them.

Keep the project's public `README.md` about the program: what it does, how to build and run it, its original sources and licenses, and its known limitations. Readers do not need the history of your local folders or the chat that prepared them. The starter folder's README explains the templates; replace it with your program's introduction when appropriate.

Copy an editable first prompt

After saving a connection in VCC Plus's assistant panel, choose **Copy Project Setup Prompt**. Paste it into your project chat and add your completed `START_HERE.md` request. **Edit...** opens the starter text before copying. You can change the language, controls and working rules to suit your project, or choose **Reset to Starter**. Your edits remain while that connection panel stays open, including when you close and reopen the editing sheet. Keep a durable project-specific version in your project documents.

The copied text includes the current connection name, installed helper path and exported connection-file path. It does not include the file's contents. Copying only changes the clipboard. Review the prompt before sending it: the setup permission takes effect when you give it to the assistant.

The permission lets the assistant register, check and restart the named VCC Plus connection through supported setup tools and make read-only requests to verify access. It stays within the permissions you exported. It does not create a new grant, authorize a different build recipe, or give access to unrelated projects. If a required native permission is missing, the assistant should explain the particular choice you need to make in VCC Plus.

The plain-text starter contains the same working rules without your connection details. It is also useful when preparing a project before opening its workspace. **Copy First Prompt** remains the smaller choice for checking just the current editor connection.

Check three kinds of access

A project connection and a documentation connection are not always separate servers. BASIC, C and assembly connections include Manual Library tools. Archive Search has its own read-only connection, exported from the Archive Library's connection panel. Verify the three kinds of access independently:

Scope	Ask for evidence
Project	Read the actual shared title and revision, using the advertised project tools.
Manual Library	Use `search_help`, then `read_topic` on one returned topic.
Archive Library	Search through its advertised tools and retrieve a relevant citation or document.

A configured server, tools visible in this chat and a successful request are three different facts. Record them separately in `MCP_SETUP.md`. If the project works but Archive tools are unavailable, continue project work that does not need the Archive while repairing that connection. Do not conclude that VCC Plus lacks a feature just because a chat has not discovered it.

Keep exported grants outside Git and out of messages. A setup record can say which workspace and permissions were checked without copying private connection files. The connection chapter explains supported reconnection and direct helper checks.

Use the local reference shelf first

The installed manuals describe the environment you are actually using. Start there:

1. Read the relevant VCC Plus BASIC or assembly manual, or the C API reference. Prefer the documented workspace workflow and bundled APIs.
2. For compiler switches, assembler directives and linker options, consult the complete CMOC and LWTools references. They retain the authors' manuals and identify the matching tool versions. The accompanying command-line help records options offered by the bundled tools, including options not covered in the upstream prose.
3. Use Archive Library for historical manuals, magazines and original hardware material. Preserve the publication and page citation.
4. If these sources leave a gap, describe it and consult the original author's website or another appropriate primary source. Record which version the answer covers.

For example, an unfamiliar CMOC switch is a reason to search the bundled compiler manual and help, not immediately change the Makefile to match a web example. An old magazine may describe another ROM, processor or video mode. If the installed API, manual and observed program disagree, record that disagreement and investigate it. None of them should silently disappear from the explanation.

Choose a machine you can prove

Write memory sizes in **KB** and distinguish the guest processor from the Mac's processor. A useful target might read: “CoCo 3, MC6809, 128 KB, keyboard and joystick, disk boot.” A CoCo 2 edition could have its own 64 KB target, colors and acceptance checks. These are different builds or configurations until you prove otherwise.

Record the official installed VCC Plus version and build agreed for the project. Application projects normally treat the emulator as a black box: public MCP tools, the manuals and documented APIs are available; changing the emulator or using its private internals is outside that scope. Name any additional source examples or artwork the owner has approved as references.

A successful 512 KB test does not establish 128 KB compatibility. Keyboard input does not establish joystick support. If the available automated run cannot select the required machine, mark that check pending and use an approved compatible test environment or owner-assisted check. Do not change the requirement to make the test pass.

For a joystick project, the setup prompt and templates ask for separate input acceptance. Discover the current controls, then use owned joystick input for the required port, axes and fire buttons. Include neutral/release intervals and pause/menu behavior; compare delivery traces with actual movement in captured frames. Shared input diagnostics distinguish controller preview, last submission and actual core values. An emulated analog test does not qualify a physical gamepad or high-resolution interface. If a small C reproducer is needed, run the compiled target at the project's optimization level as well as any host tests.

Begin with one runnable slice

For multi-disk software, plan the owned disk-exchange tests as part of the project brief: which disk boots, which drives hold Home and Data, where a one-drive fallback asks to swap, where saves belong, and how wrong, empty, corrupt and write-protected game data should behave. Declare every test image as a reviewed build product. Verify independent writable copies, per-drive save/read activity, save/reload across swaps and explicit retained-set resumption. Match the program's actual drive choices, including nonzero Home drives; a successful mount alone is not an acceptance test.

Ask for the smallest program that proves an important part of the experience. For a game, that might be one object you can move and stop. For a disk utility, it might be listing one known directory without altering the disk. Agree on the expected result before running it.

For a C project, start with standard CMOC functions and the documented libraries. Ordinary application logic belongs in C. If the requested experience seems to require assembly, direct hardware access or drawing outside those APIs, first measure the actual limitation. Put the proposed exception in `DECISIONS.md`: the missing capability, evidence, alternatives, smallest change and checks needed. The owner or named reviewer decides whether to approve it. An assembly project naturally starts with its agreed instruction set and hardware scope; it does not need a separate exception for every assembly instruction.

Use the project's approved build recipe. Test a source revision and its resulting artifact together. Compilation proves that a program was accepted by the tools; playing it, using its controls and checking its output establish its behavior.

Prove which physical code overlay is loaded

A packed game can replace code in the same physical RAM, rather than merely switching between different RAM banks. Discover the installed C connection's `get_capabilities` first: `banked_debug_manifest_versions` includes `2` and `verified_code_overlays` identifies this support. Use the approved Make workflow and configure `debugManifest` to name the build-produced evidence. The project needs its own source and recipe, the installed tools and public MCP interface; it does not need the emulator's private source tree.

Manifest version 2 adds explicit `overlayGroups` and an `overlay` identity on each member module. This structural fragment shows two code owners reusing the same physical page. It is **not a complete manifest or runnable project**: keep the required entry point, binary/disk/linker identities, complete source set, artifact hashes and remaining modules in the real manifest.

```
{
  "version": 2,
  "overlayGroups": [{"id": "stages", "owners": ["stage-a", "stage-b"]}],
  "modules": [
    {
      "id": "stage-a-code",
      "sections": ["stage_a_code"],
      "codeSections": ["stage_a_code"],
      "image": {"path": "build/debug/stage-a.raw", "logicalAddress": 16384},
      "pages": [{"logicalPage": 2, "physicalPage": 30}],
      "overlay": {"group": "stages", "owner": "stage-a"}
    },
```

```
    {
      "id": "stage-b-code",
      "sections": ["stage_b_code"],
      "codeSections": ["stage_b_code"],
      "image": {"path": "build/debug/stage-b.raw", "logicalAddress": 16384},
      "pages": [{"logicalPage": 2, "physicalPage": 30}],
      "overlay": {"group": "stages", "owner": "stage-b"}
    }
  ]
}
```

An owner may contain several modules. **All of that owner's code-bearing modules must be mapped concurrently and every executable guard byte must match the current RAM snapshot.** Exactly one matching owner establishes residency. Zero matches, partially installed code or multiple matching owners withhold that group's source locations, functions and variables. A familiar address or matching first opcode is insufficient. Different groups and ordinary resident modules cannot claim the same physical storage; same-owner modules cannot overlap it.

Keep executable bytes immutable in `codeSections`. Mutable data and BSS belong in separately declared sections/modules with the same group/owner identity. They inherit the uniquely proven code owner rather than being compared with old initial values; their storage must still have the declared current RAM mapping. A data/BSS-only owner cannot prove residency, and BSS needs no invented raw zeros.

Hashes still bind the complete declaration to its retained build, with independent object/relocation verification. Bounds remain explicit: 1 MiB manifest, 32 MiB per compiler `.s` file, 8 MiB per other evidence file and 128 MiB aggregate evidence; at most 64 groups, 256 owners and 256 modules. Group/owner identities are at most 128 UTF-8 bytes and contain no NUL. Declaring an overlay does not relax these checks or grant memory/register edits.

At verified owned stops, inspection reports the current module and `residency_fingerprint`. Source stepping and cached inspection receipts check fresh residency; the previous owner is not retained after code replacement. Test the real loader's A-to-B-to-A transitions, partial/wrong data and required processors using ordinary guest execution. Do not write guest RAM or registers through a debugger to manufacture a successful load or test.

Keep results another person can use

For each completed stage, record the source revision, build options, artifact name, target configuration and observed result. Use writable test copies of media. Obtain an idle-desktop window before automation that moves the user's pointer or types. Use direct MCP captures where supported; a screenshot of the CoCo display is often more useful than the surrounding Mac interface.

When measuring speed, repeat the same action on the same configuration and state the units. A redraw time, emulated cycle count and wall-clock startup time answer different questions. Keep the baseline result and any remaining limits alongside the improved result.

Finish with a reader-facing README, reproducible source, the agreed runnable artifact and useful screenshots. Preserve original attribution and file-specific license notices. Keep credentials, private ROMs, raw personal captures and temporary logs out of the repository. Commit, push, publish or message another chat only within the owner's actual authorization; do not repeatedly ask for a routine action already authorized for the same work.

For exact build delivery, separately approve a local folder through the C Assistant panel's **Allow Build Output Export**. Ask the assistant to export the original declared Release/Debug products, retain their build/artifact identities, and independently compare all receipt hashes. It can retrieve an earlier Release set after building Debug without another build. Keep delivered build bytes separate from retained writable test disks. Deliver an existing build

For a manual demonstration, the emulator header's camera saves the actual CoCo picture using the remembered None/5/10-second screenshot timer. Selecting a timer changes the setting; clicking the camera takes the picture. The separate video control records the CoCo picture and optional generated sound to MP4 when stopped. Screenshots and videos share one remembered Save To folder, Desktop by default. Both exclude System artwork, Mac controls and the desktop. The local shutter and recording-start cues never enter the output.

Capture controls and recovery

A video-enabled MCP connection also lets your assistant record a short gameplay demonstration from its own disposable playtest. Ask it to use normal keyboard or joystick input, record the relevant route or interaction, retrieve the completed MP4 and verify its byte count and SHA-256 before adding it to your manual assets. Choose CoCo audio per clip. Mac windows, microphone and capture feedback sounds are excluded. Check the recording's field count and truncation status so the description matches what the clip actually shows. Waiting between tool calls adds no footage. These tools serve any project; they require no game-specific support in VCC Plus. Inspect live capabilities before requesting a recording.

A local working disk can also become a **Personal Library** entry without uploading or publishing it. For wider distribution, the software creator can deliberately choose **Library Actions → Submit Software…**, review the permission terms and offer the package for editorial inclusion. Include the required notices and obtain permission for every supplied asset; adding a private entry is not permission to redistribute purchased software.

Private imports and public submissions

Before handing the project to another chat, update the brief, connection checks, decisions and tests. Then ask the new chat to read them, verify its own access and continue from the last demonstrated working stage. That is a more reliable starting point than asking it to infer the project from a long conversation.

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