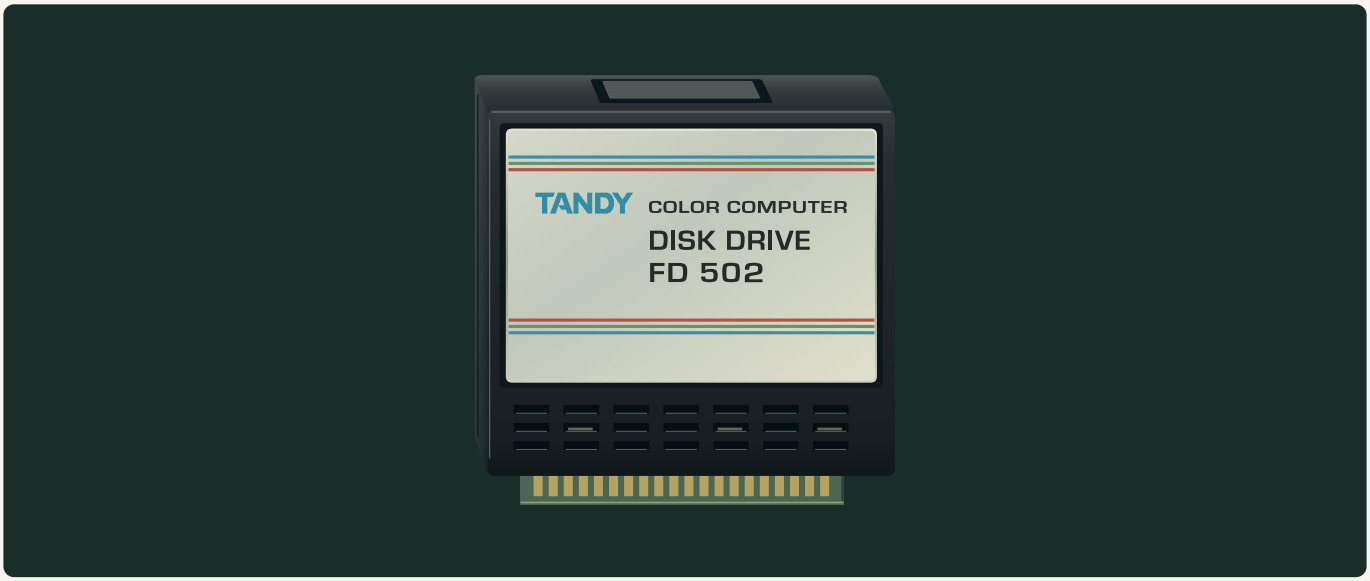
Programming in C on the CoCo 3 — CMOC and Its APIs

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?

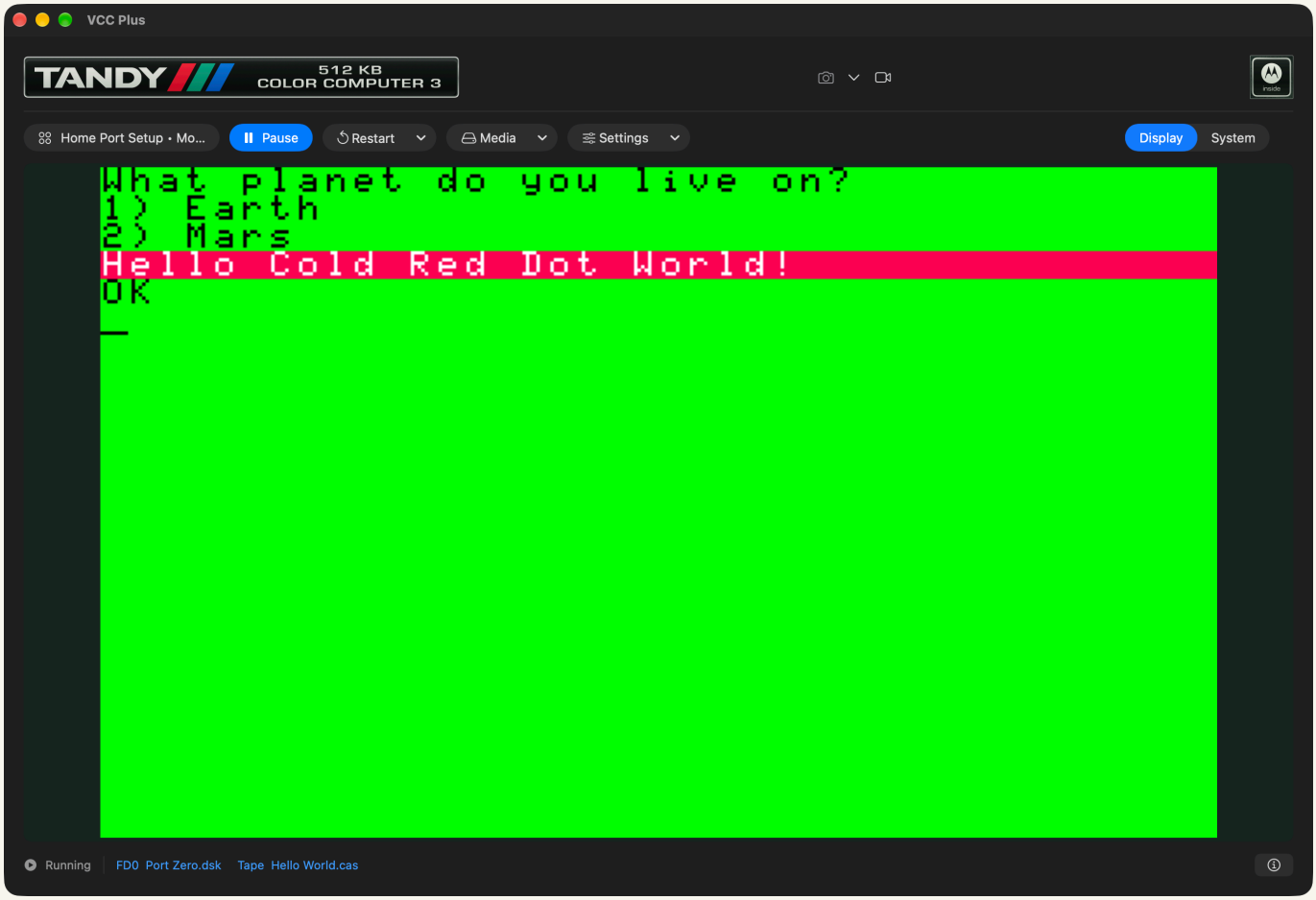


Figure 1: The running VCC Plus display after pressing 2 shows the menu, Hello Cold Red Dot World! on a red line, and BASIC's OK underneath.

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.

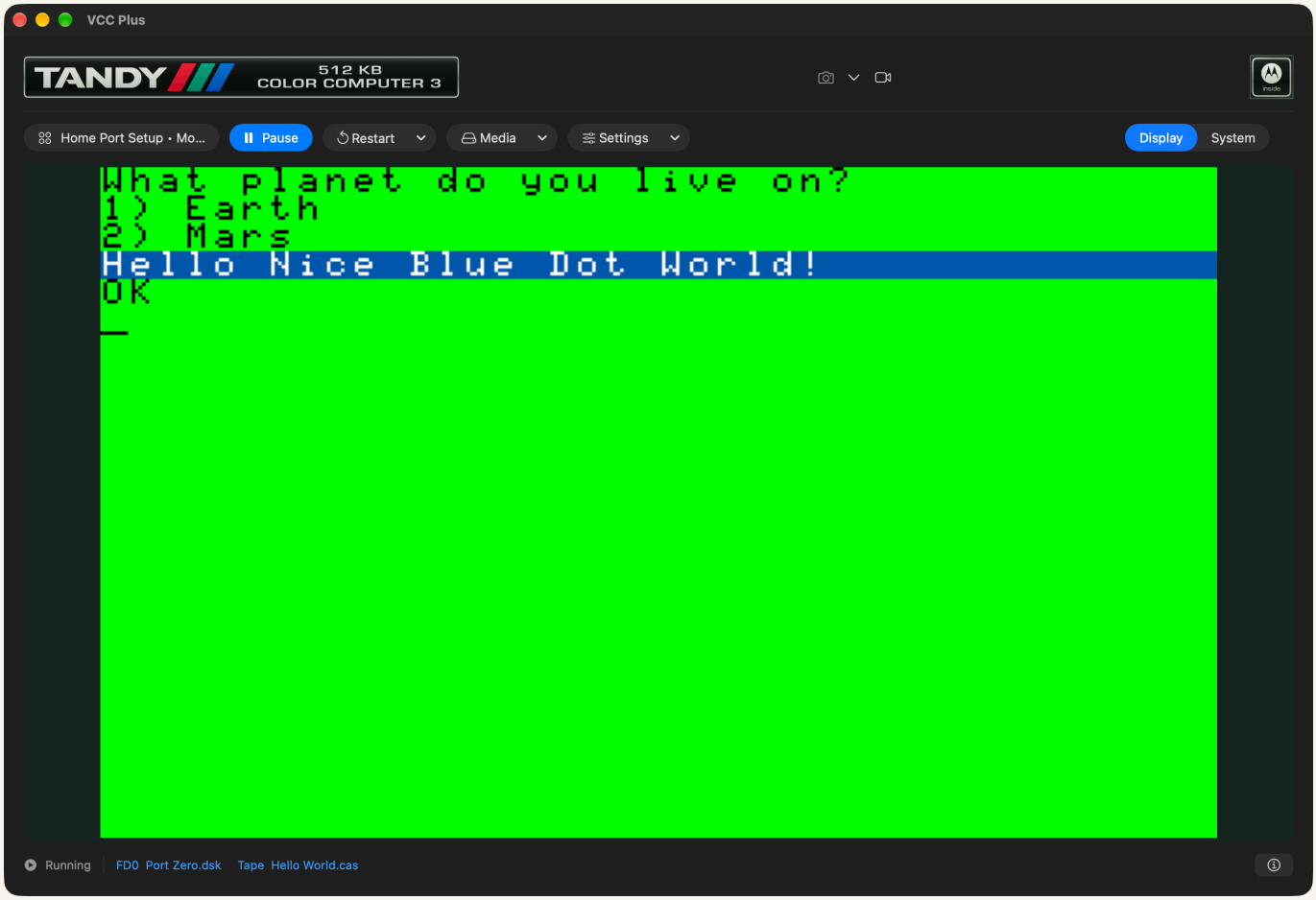


Figure 2: Earth under RGB Monitor. The greeting uses a blue band and the BASIC background is bright green.

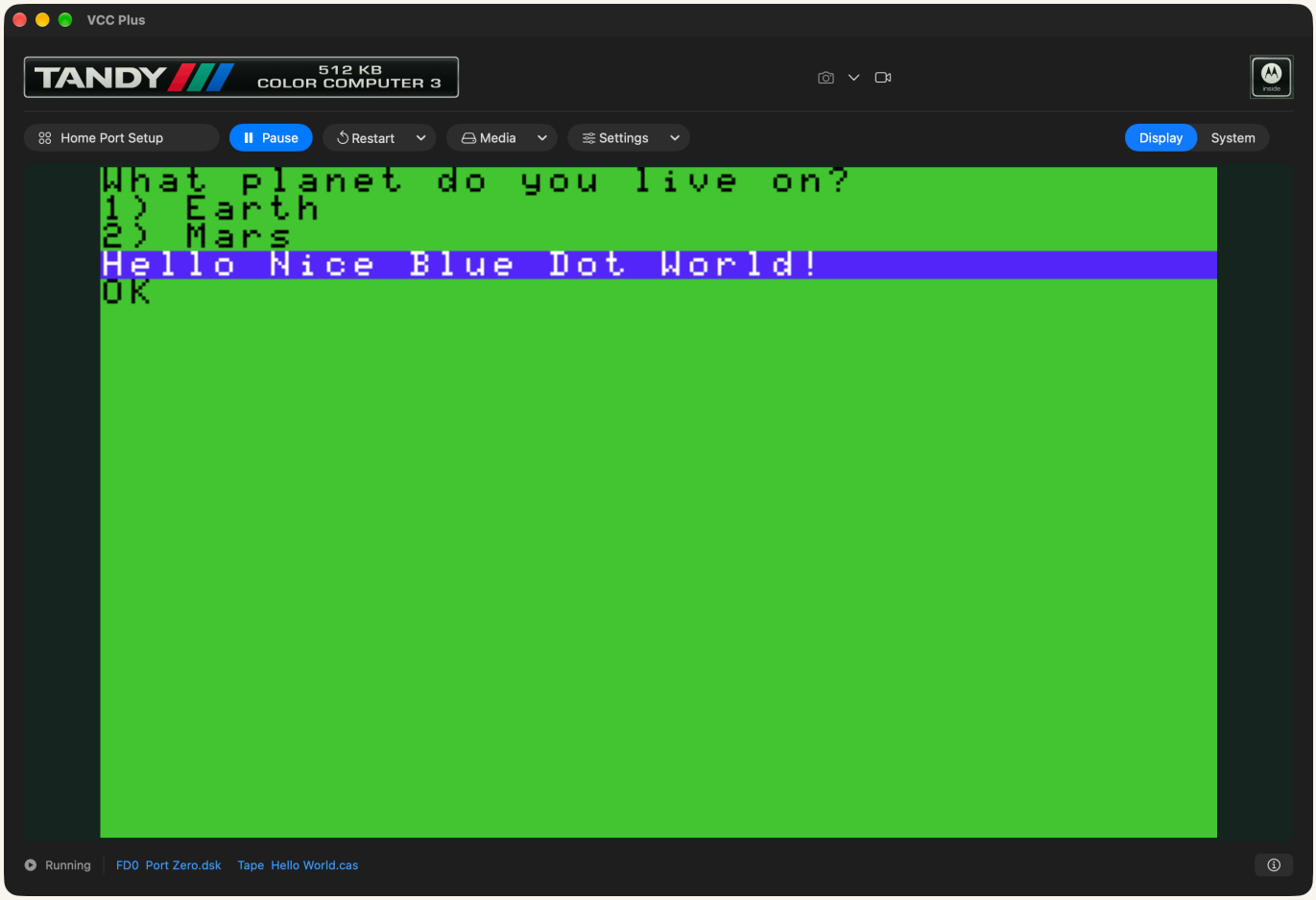


Figure 3: Earth under Composite TV with the same build, key and palette value. The band is violet-blue and the green is softer.

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.

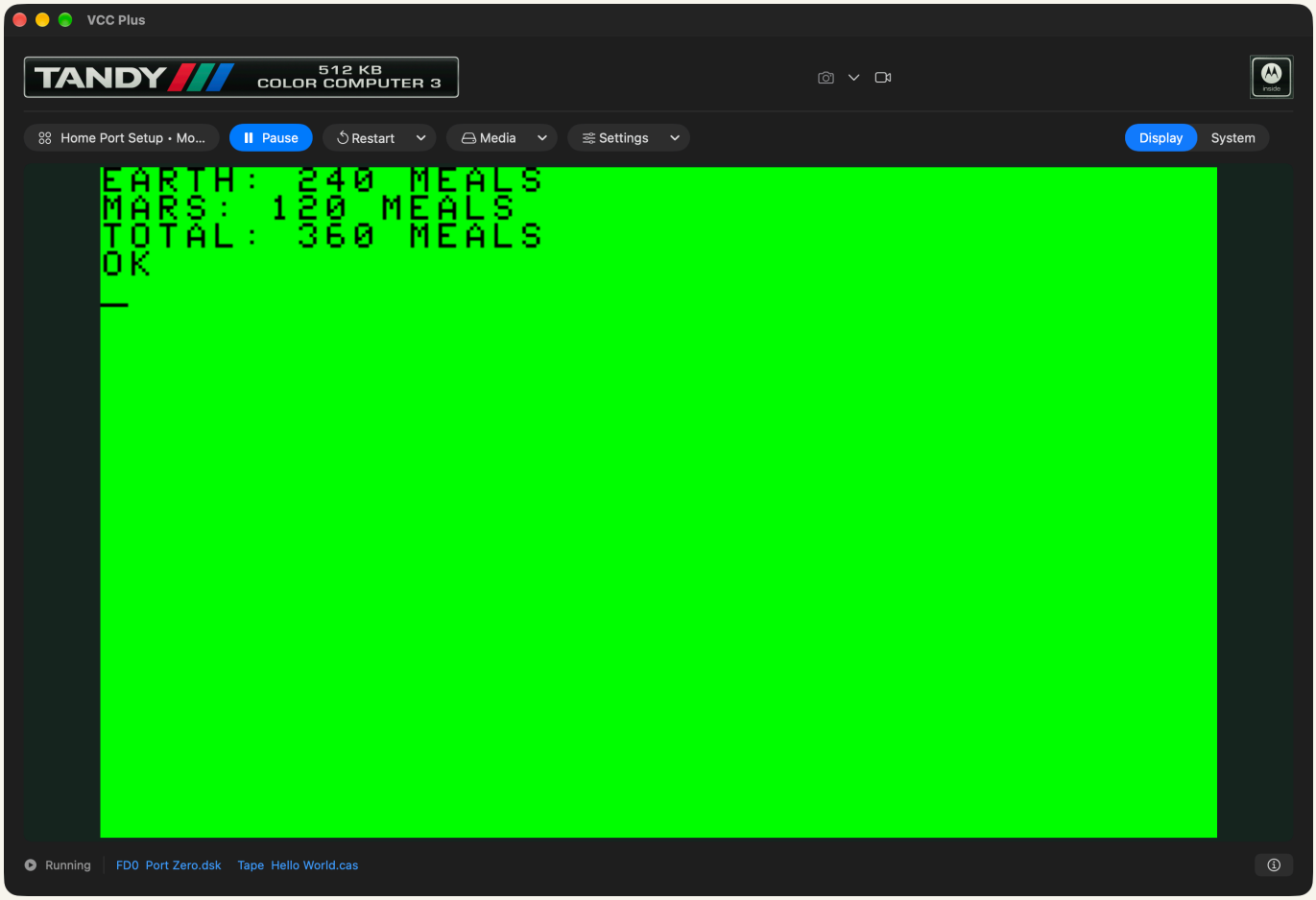


Figure 4: The real guest has printed EARTH: 240 MEALS, MARS: 120 MEALS and TOTAL: 360 MEALS, then returned to OK.

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.

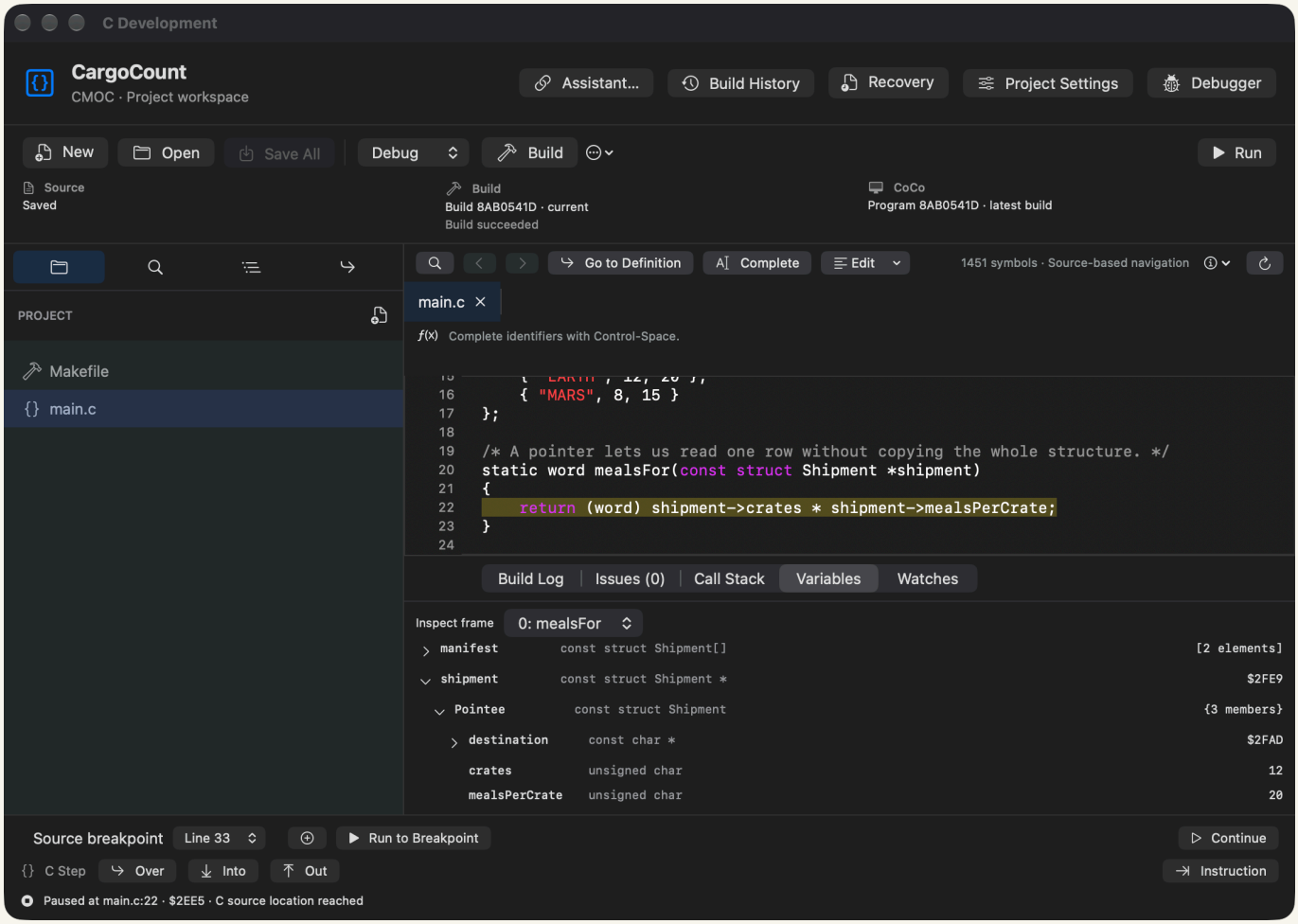


Figure 5: In the paused C Development window, line 22 multiplies the two members. Variables shows shipment expanded through Pointee to crates 12 and mealsPerCrate 20.

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**.

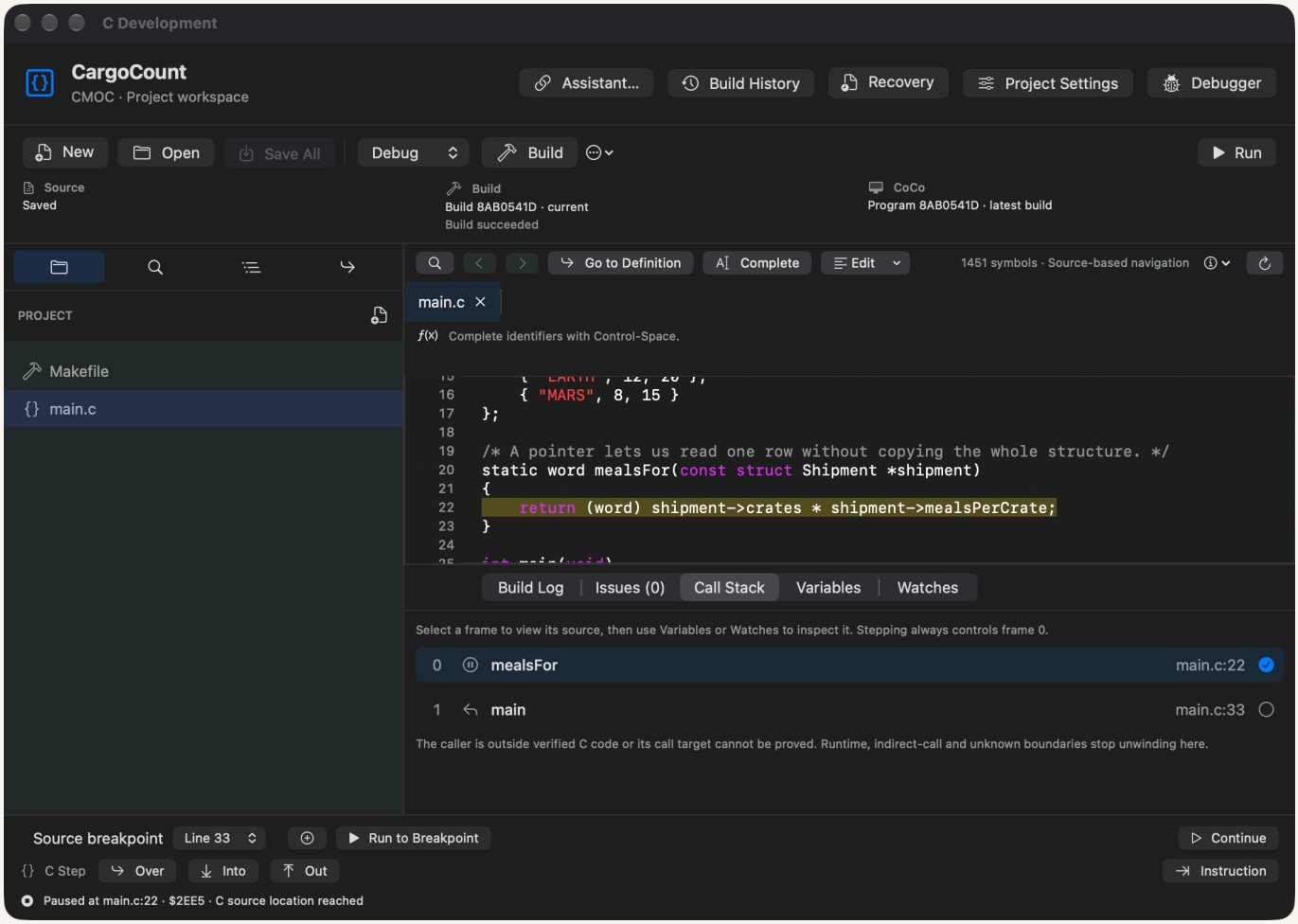


Figure 6: Call Stack shows mealsFor stopped at line 22 and its caller main at line 33.

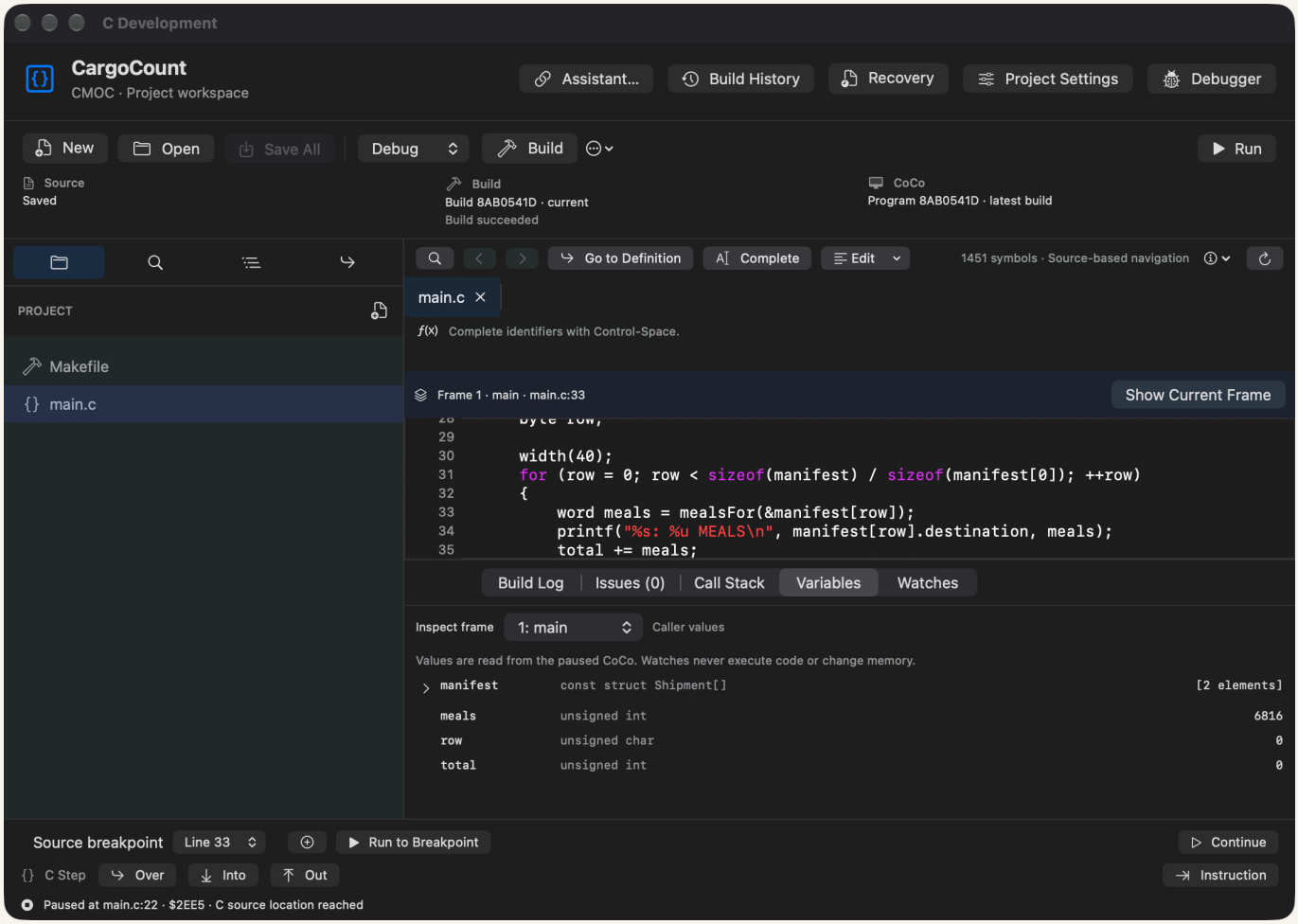


Figure 7: Selecting frame 1 shows main at its call site. The caller's row and total are both zero. Its local meals has not yet received the answer.

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.

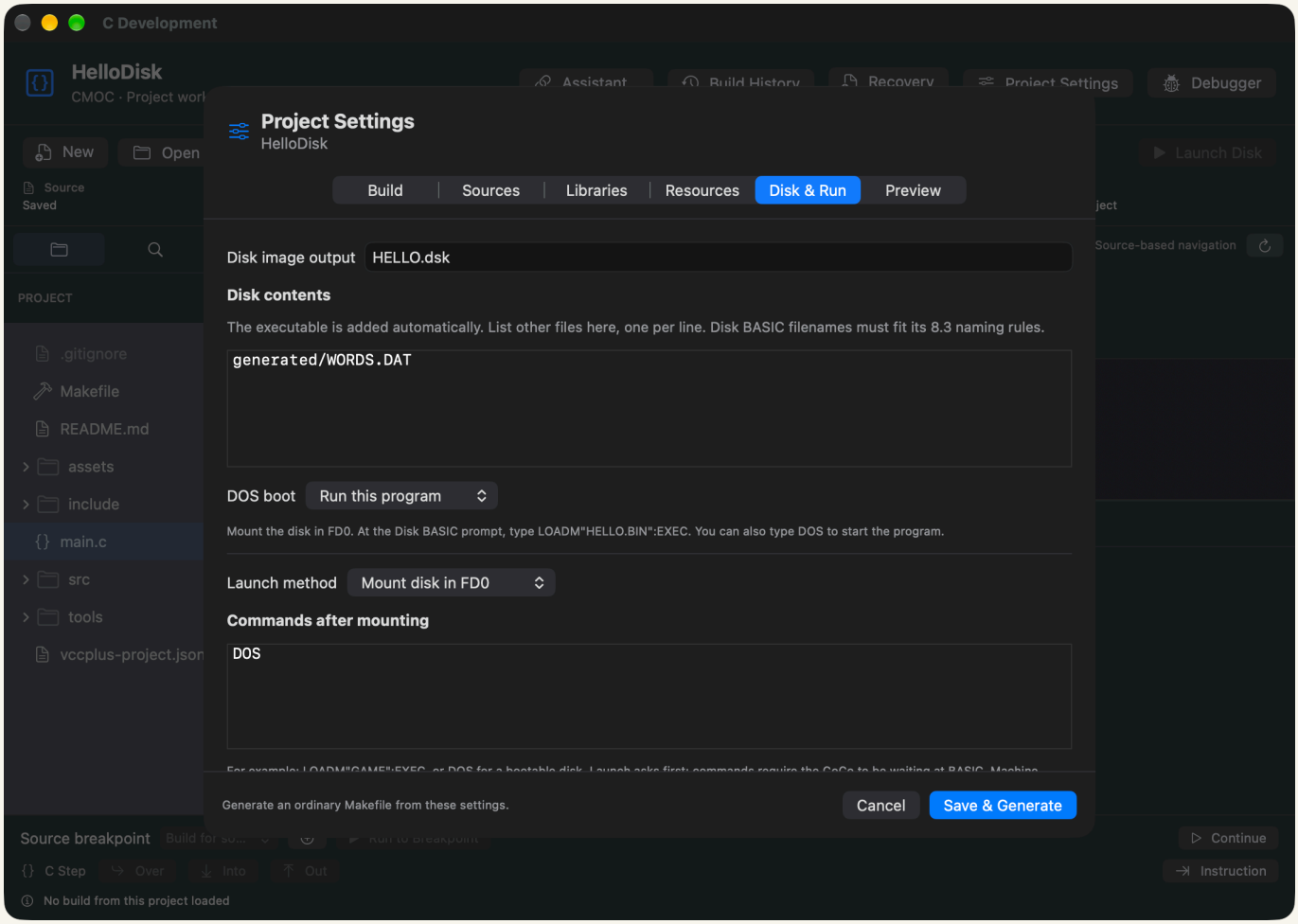


Figure 8: In the C workspace, Disk & Run lists generated/WORDS.DAT as an extra file. The executable is included automatically. The disk will occupy FD0 and DOS will launch it.

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.

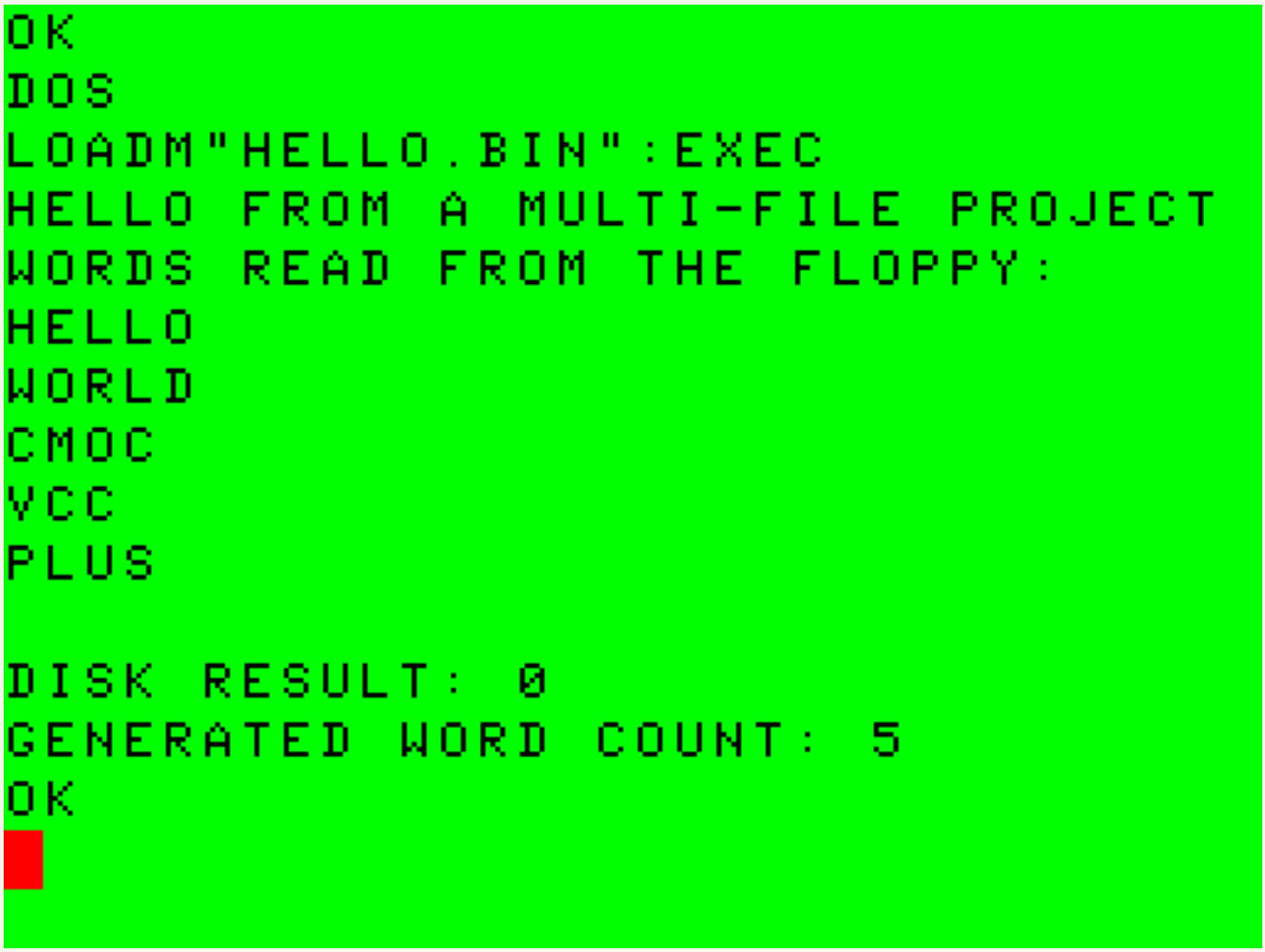


Figure 9: The running CoCo printed five words from FD0. The last two lines report the successful disk operation and the count compiled into HELLO.BIN.

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.

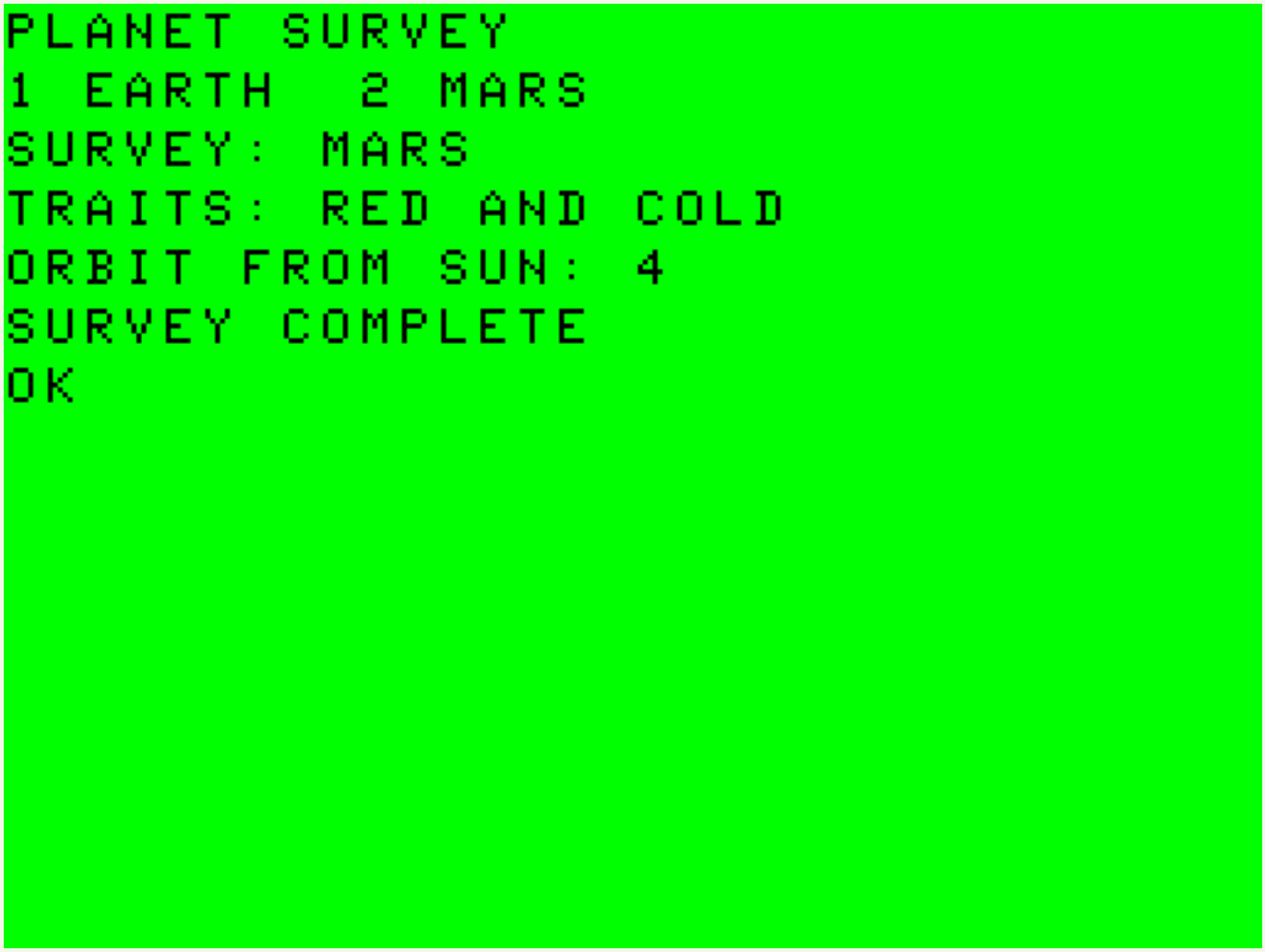


Figure 10: Planet Survey displays the two Mars traits and its orbit number after reading one disk record.

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.

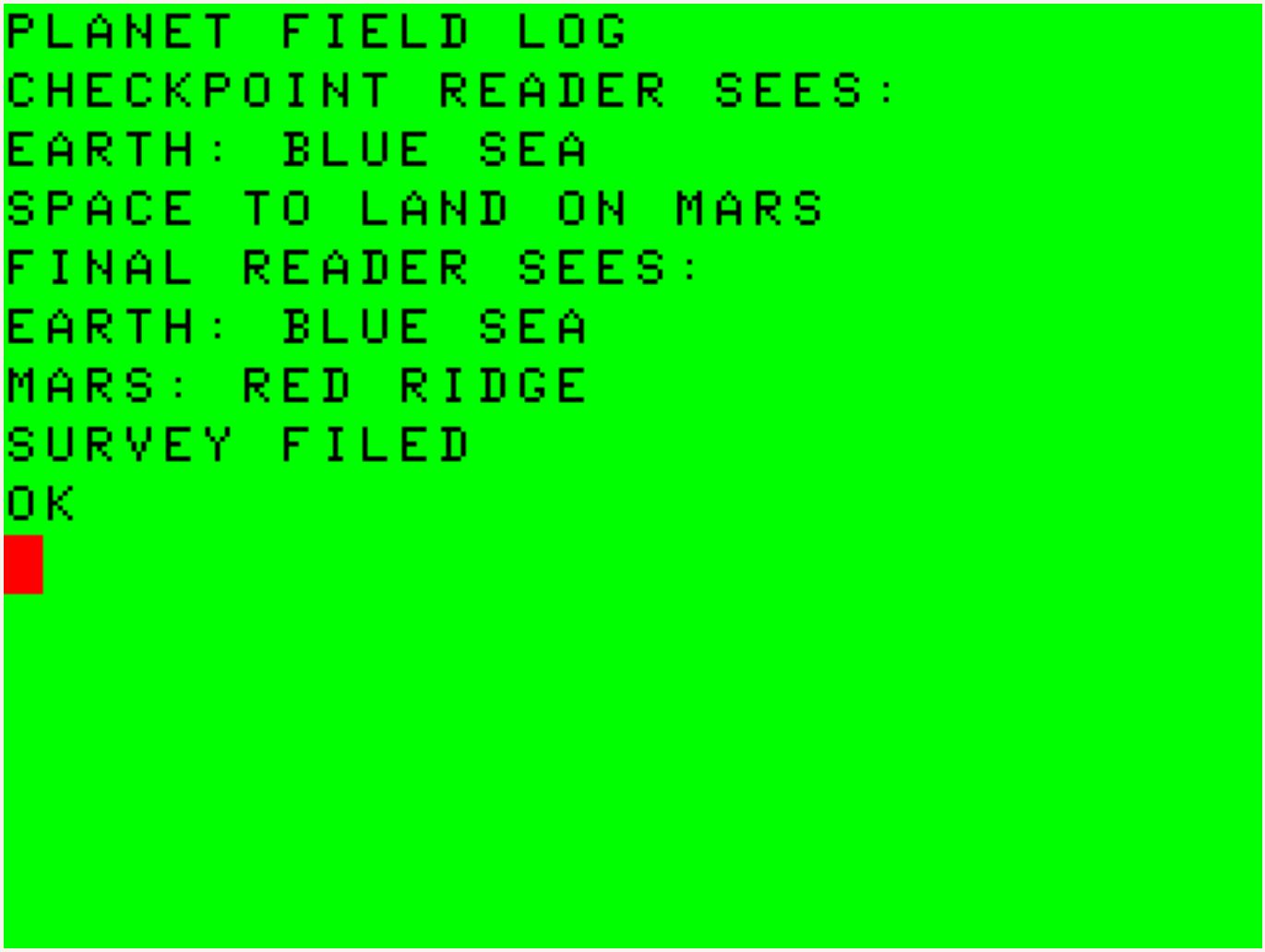


Figure 11: The running program reads the Earth report at the checkpoint, then reads both the Earth and Mars reports after closing the writer.

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.

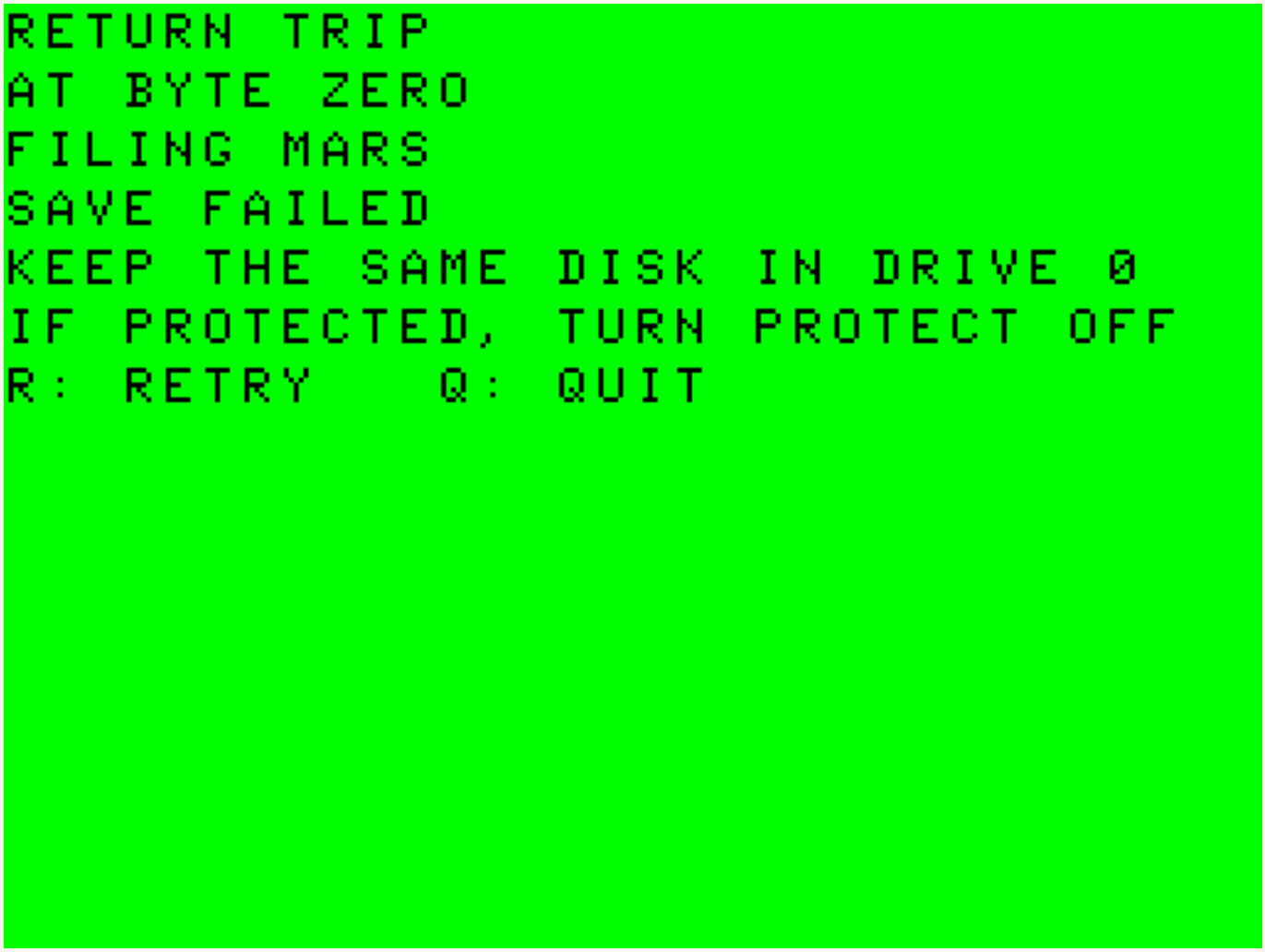


Figure 12: The running CoCo program pauses with the pending Mars save and offers a retry or quit key.

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");
    }
```

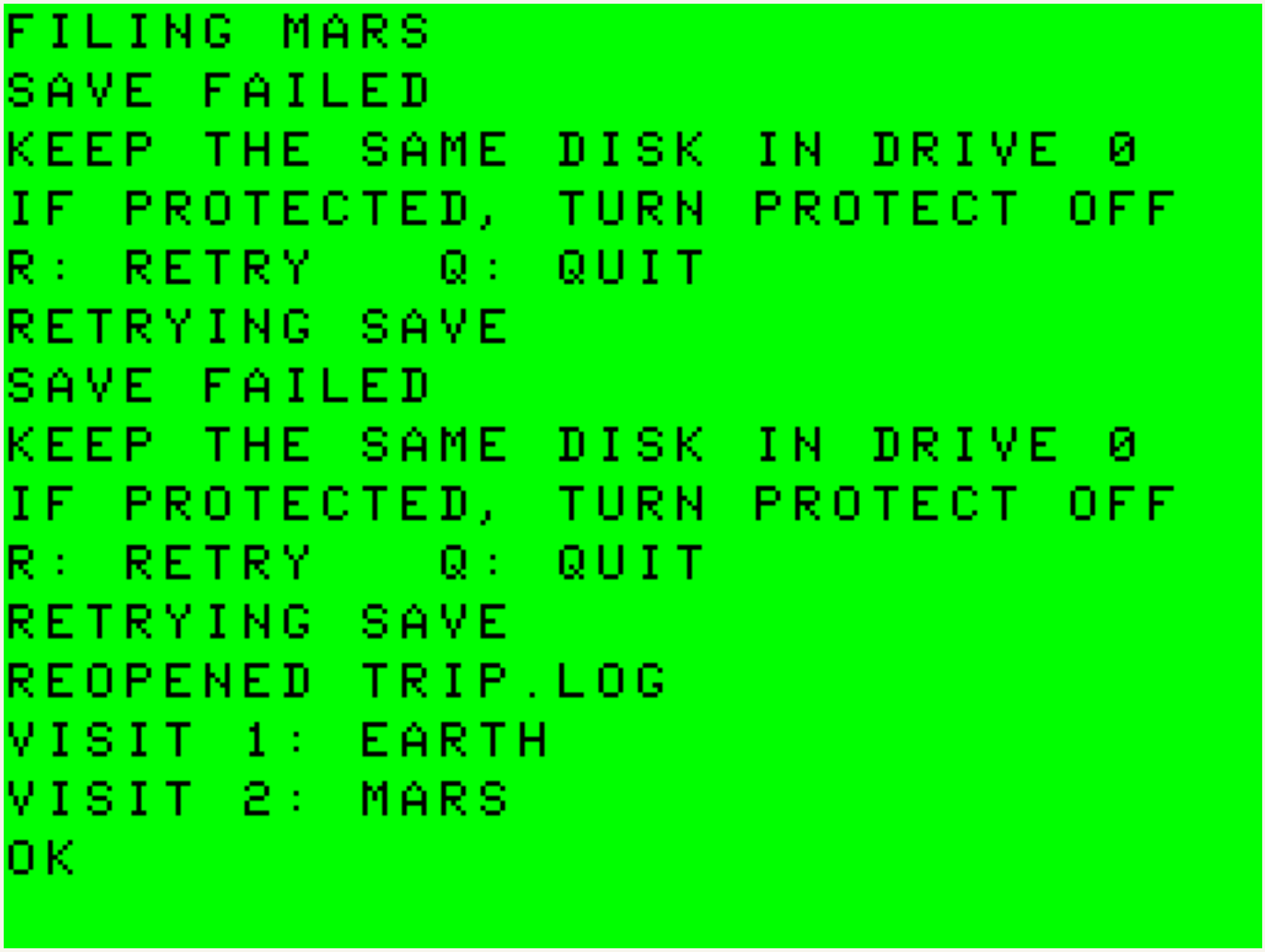


Figure 13: After protection was removed, the program reopened TRIP.LOG and displayed one visit to Earth and one to Mars.

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
```

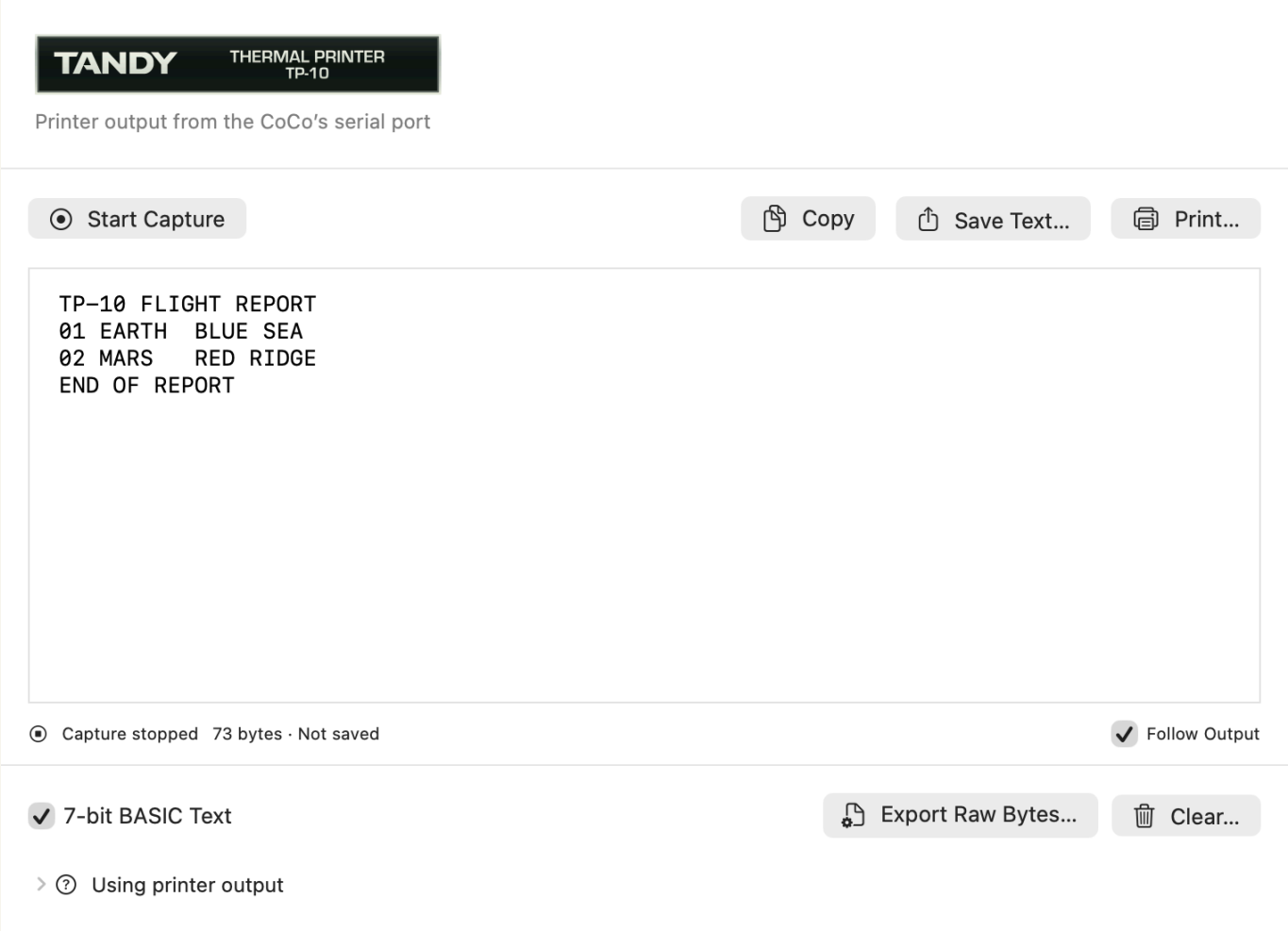


Figure 14: Printer Output retains the four-line flight report after Stop Capture. Use Save Text to keep it.

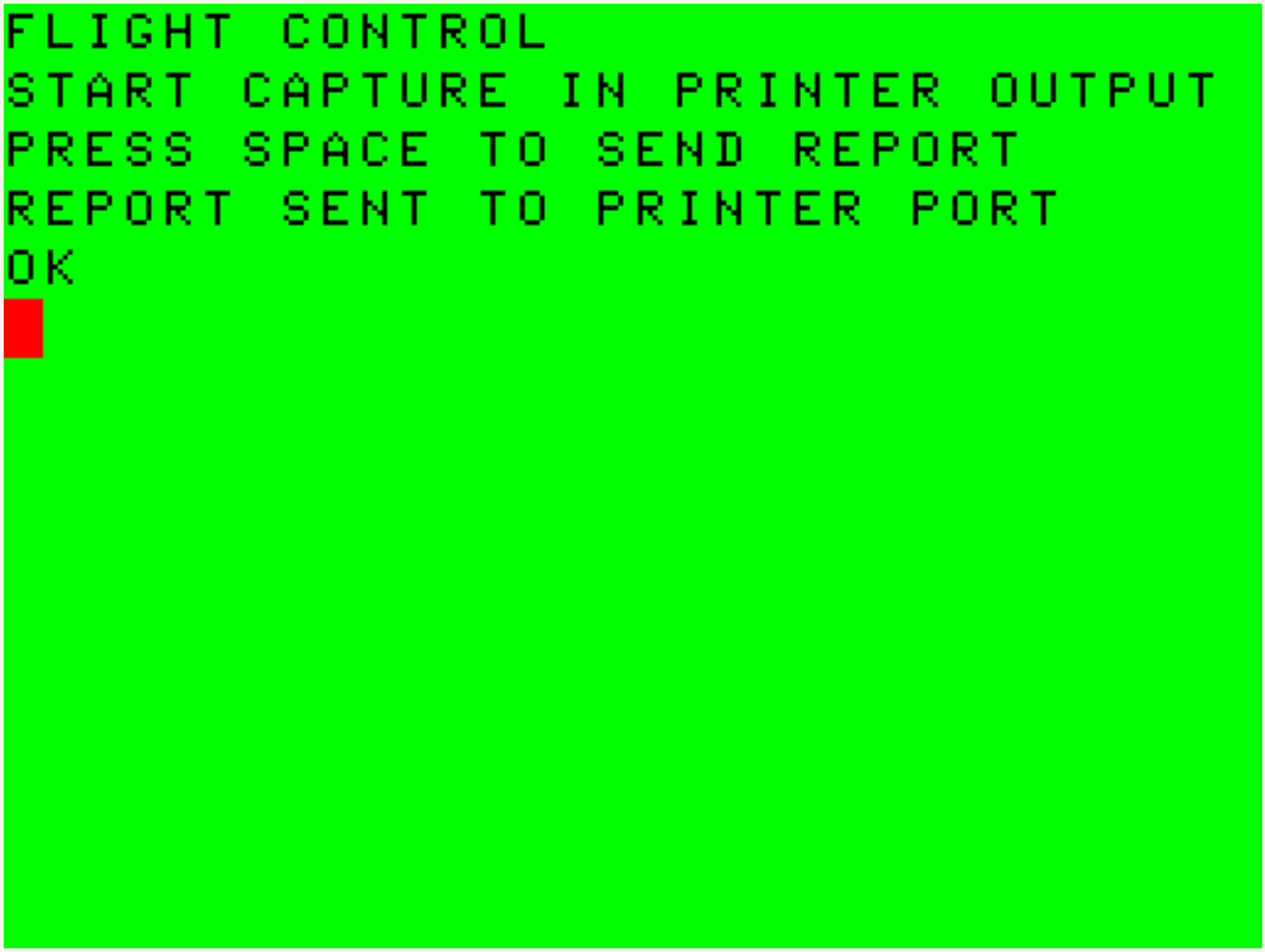


Figure 15: The CoCo shows the flight's completion message and returns to BASIC after sending the report to its printer.

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.

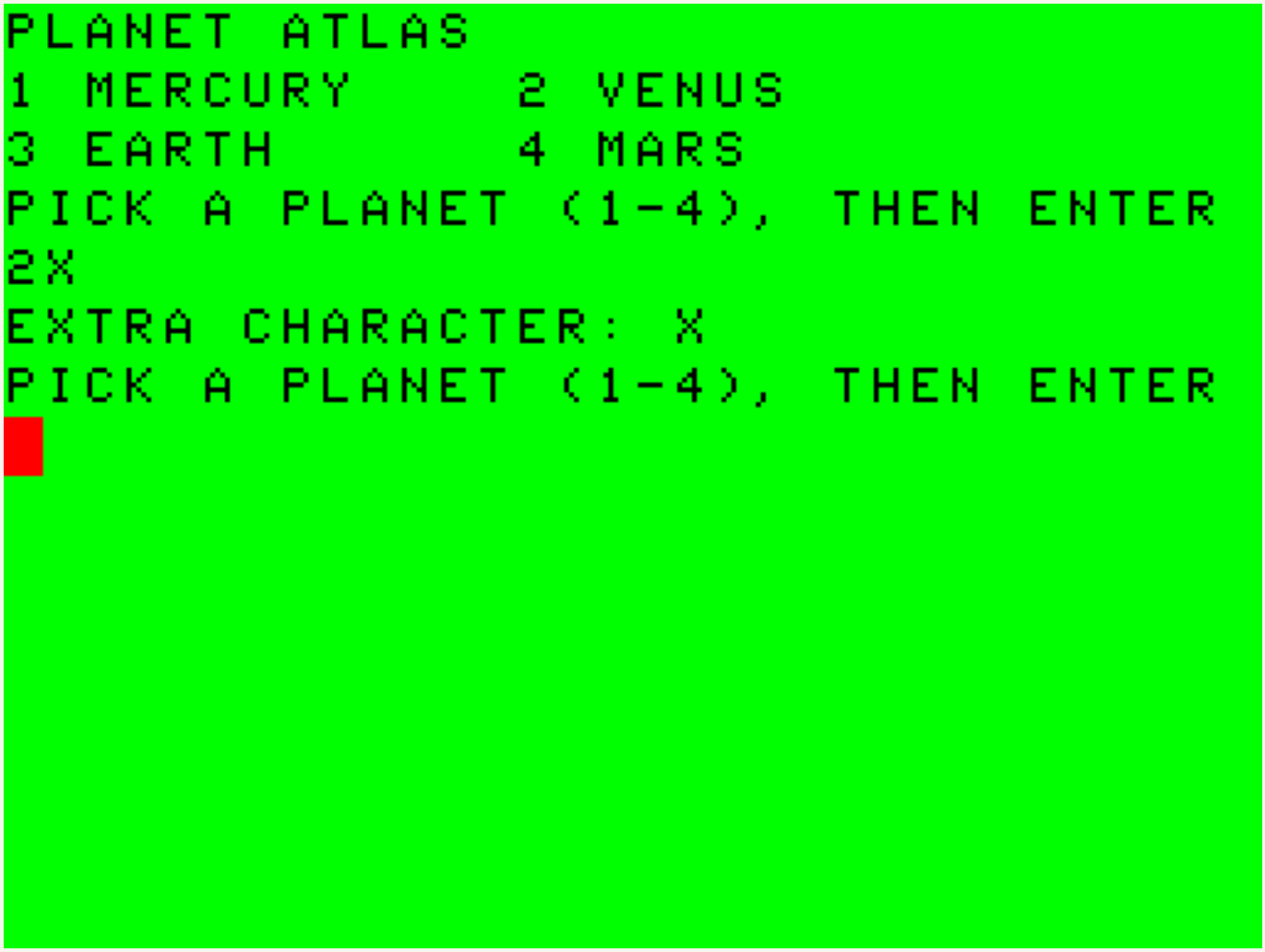


Figure 18: Planet Atlas identifies the X in 2X and asks for another destination on the guest screen.

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?

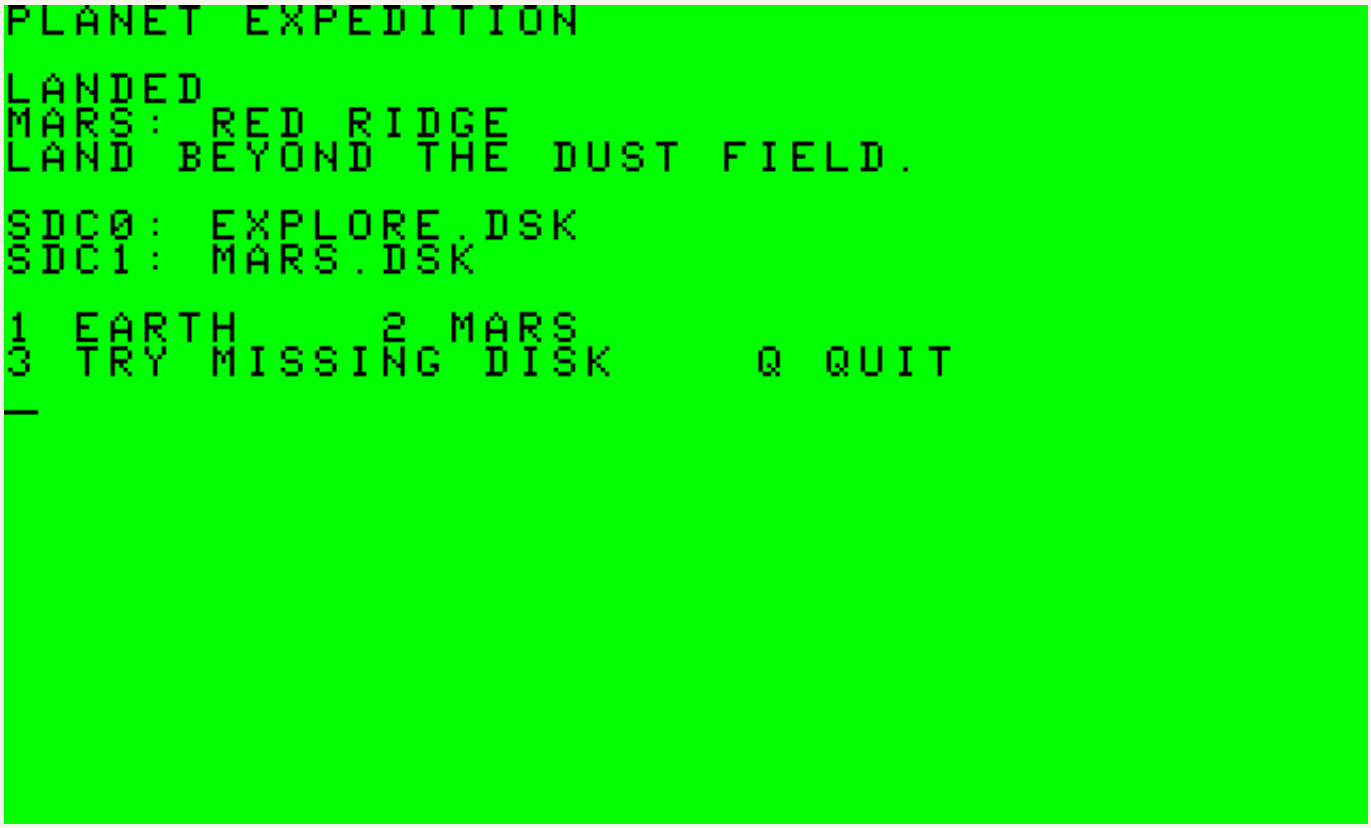


Figure 20: The running CoCo shows the Mars landing instructions. EXPLORE.DSK is still mounted on SDC0 and the Mars data disk is mounted on SDC1.

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.

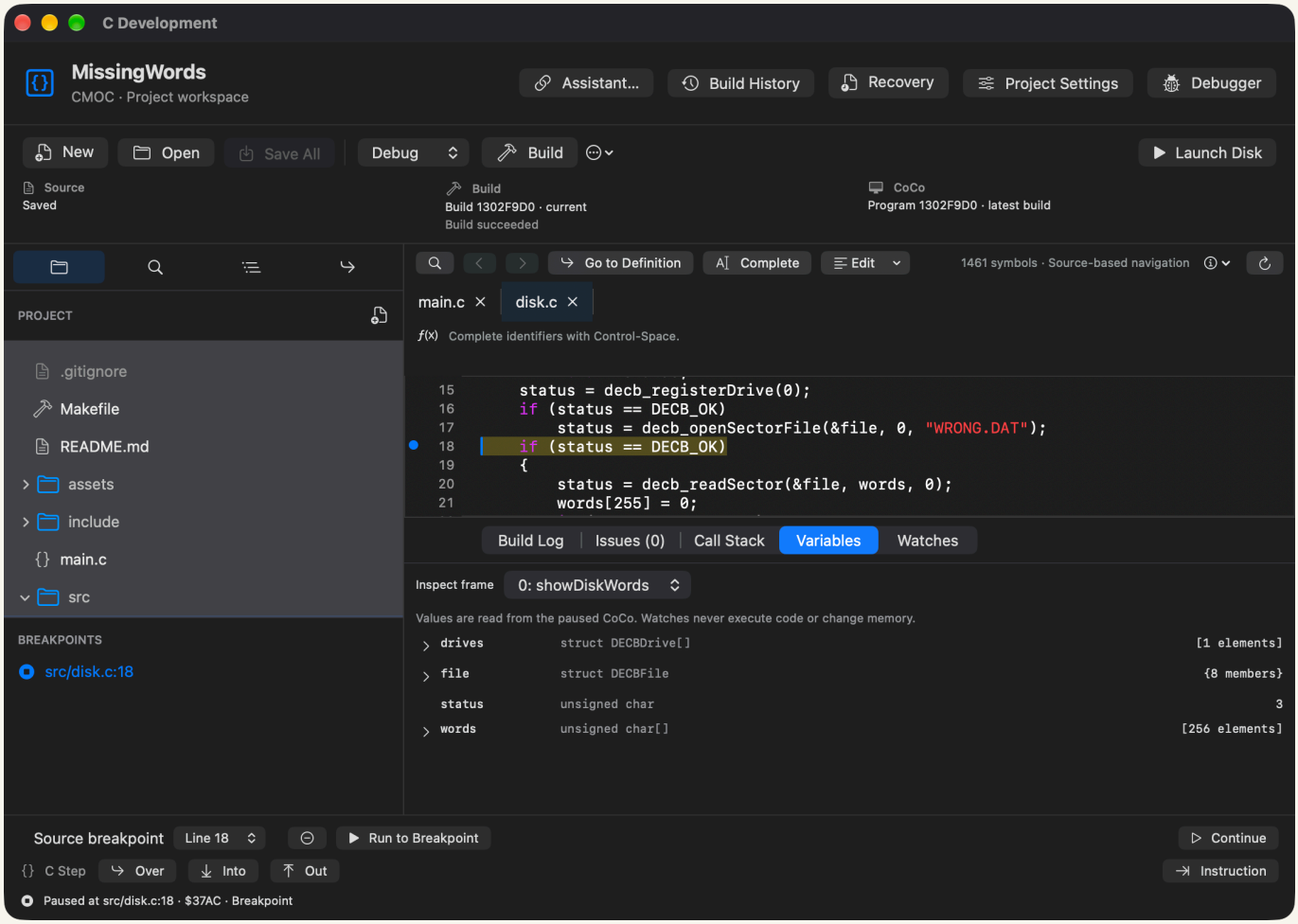


Figure 21: A private Hello Disk copy named MissingWords has stopped on line 18, immediately after the open of WRONG.DAT. In the showDiskWords frame, Variables reports status 3.

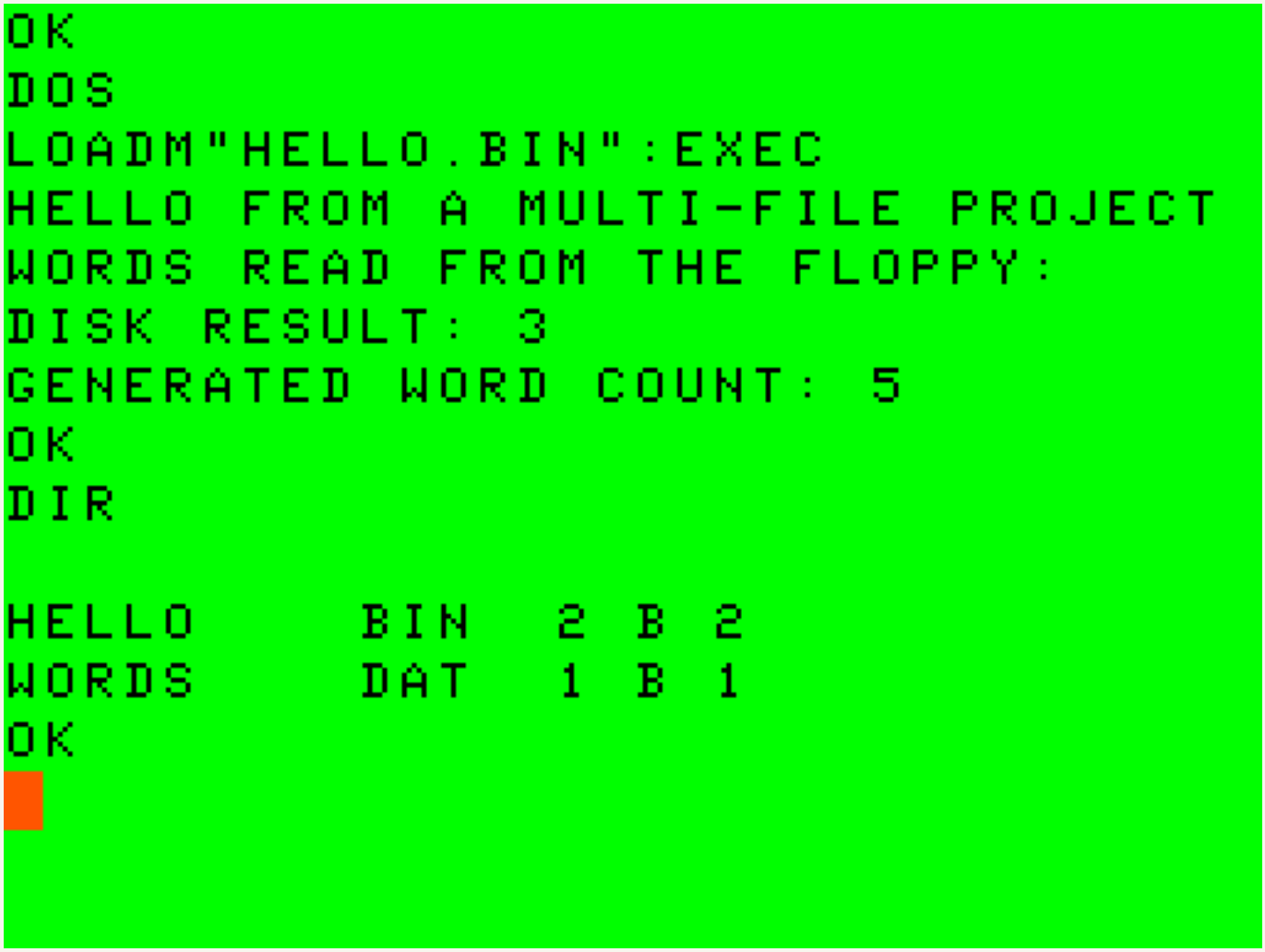


Figure 22: The same guest screen shows disk result 3 and count 5 above the DIR listing for HELLO BIN and WORDS DAT.

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?

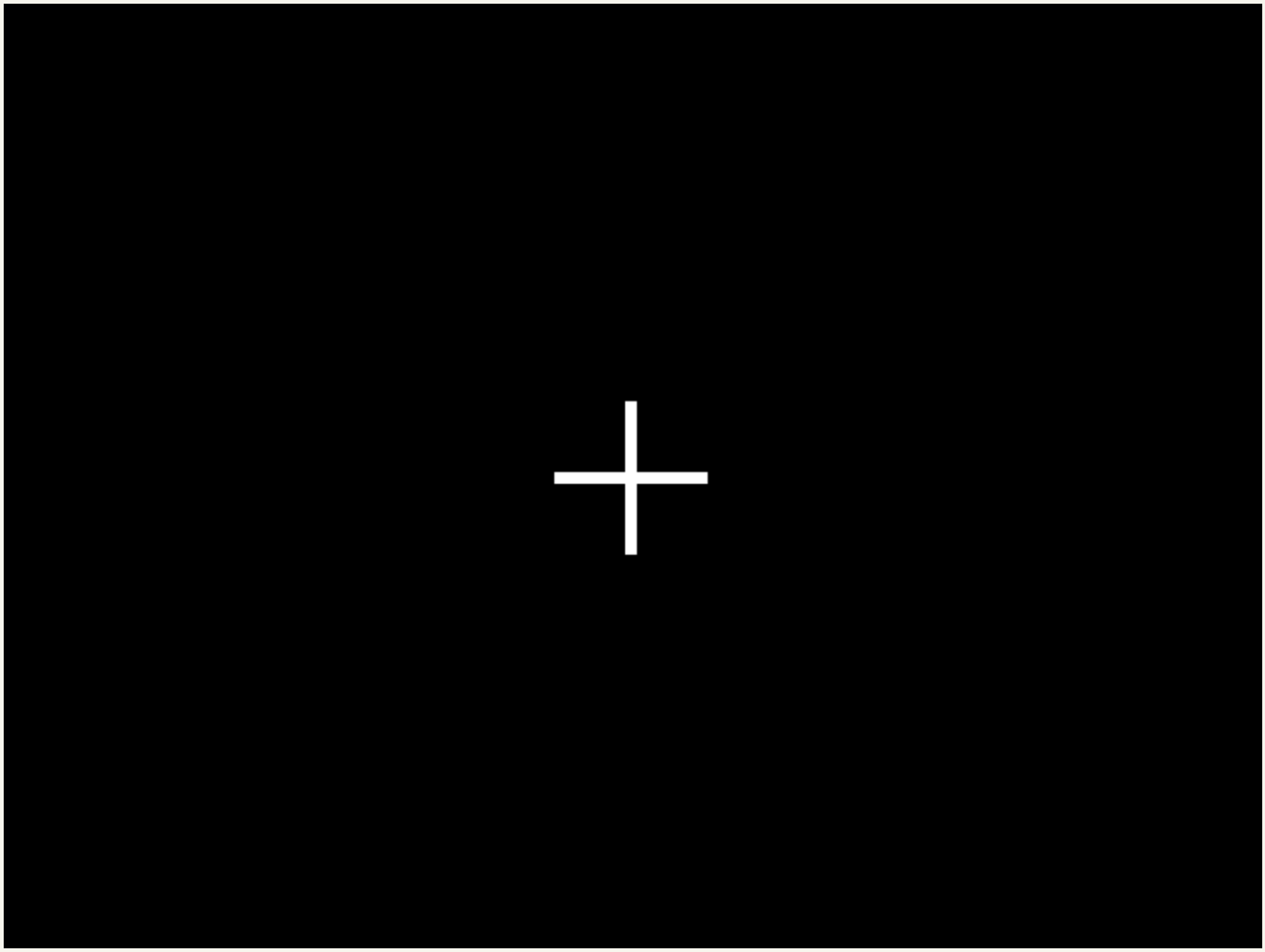


Figure 23: VCC Plus displays the white landing cross on the black CoCo 3 graphics page while the program waits for a key.

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.

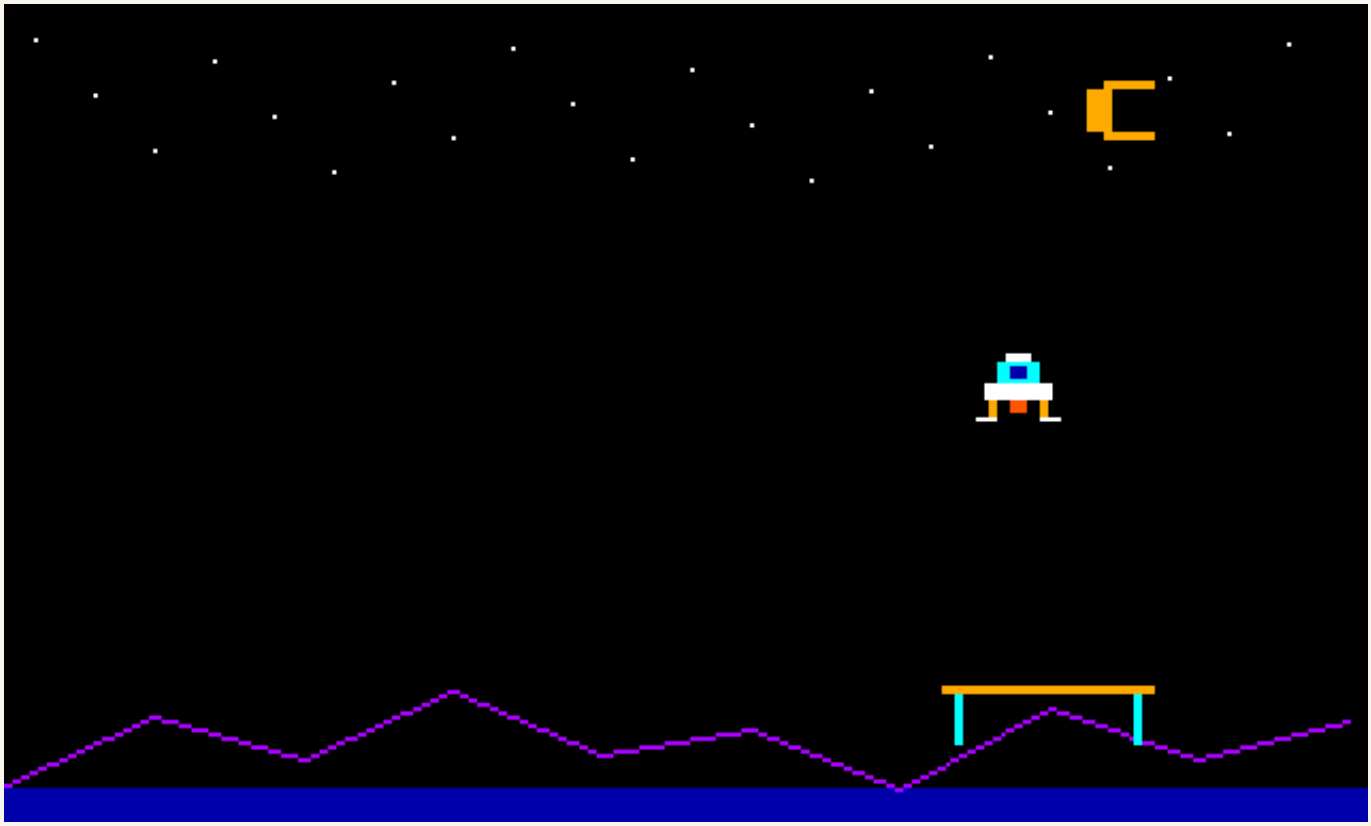


Figure 24: The running lander hovers above the gold pad on the right. The cyan legs of the pad cross a violet ridge, and a gold crescent hangs above the craft.

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.

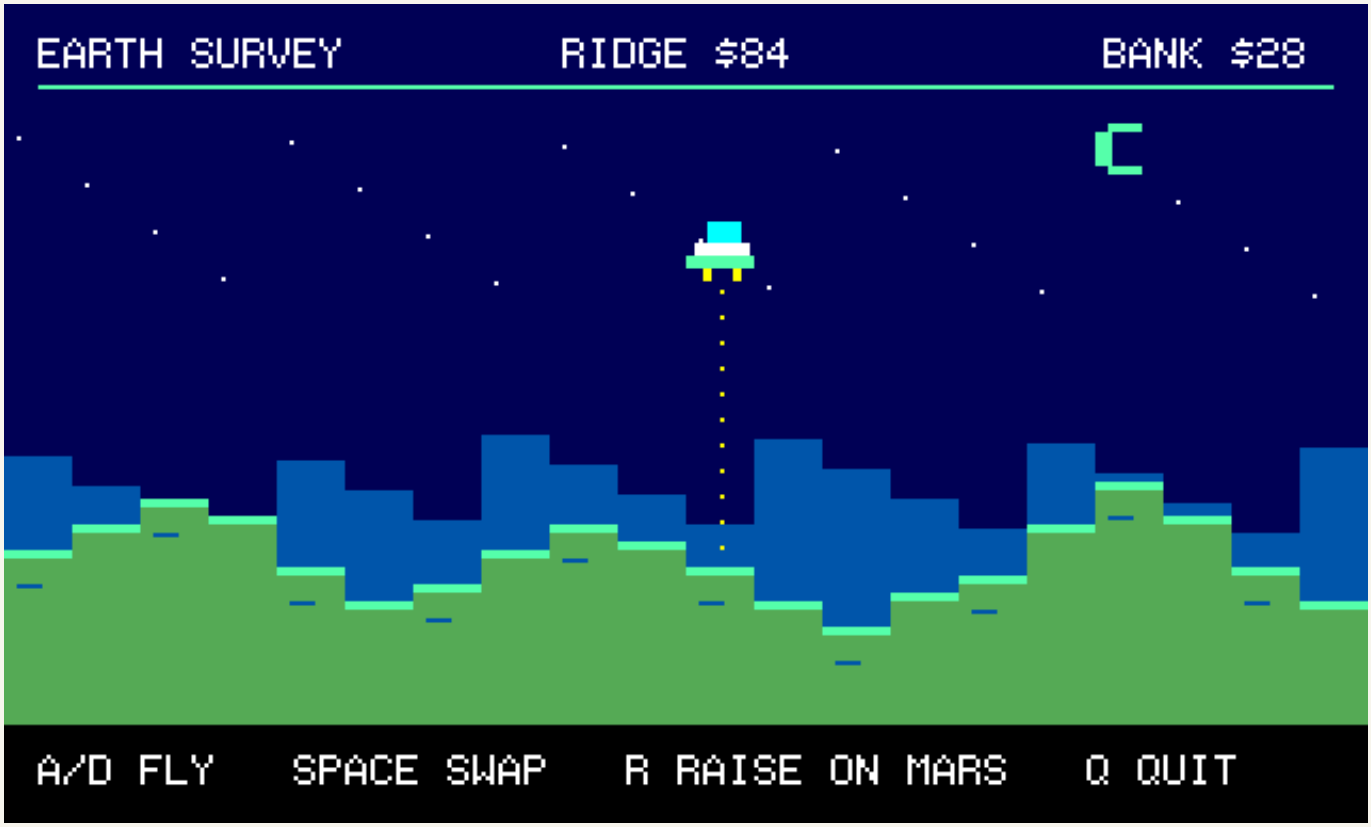


Figure 26: The probe at its starting position over Earth in the running CoCo program.

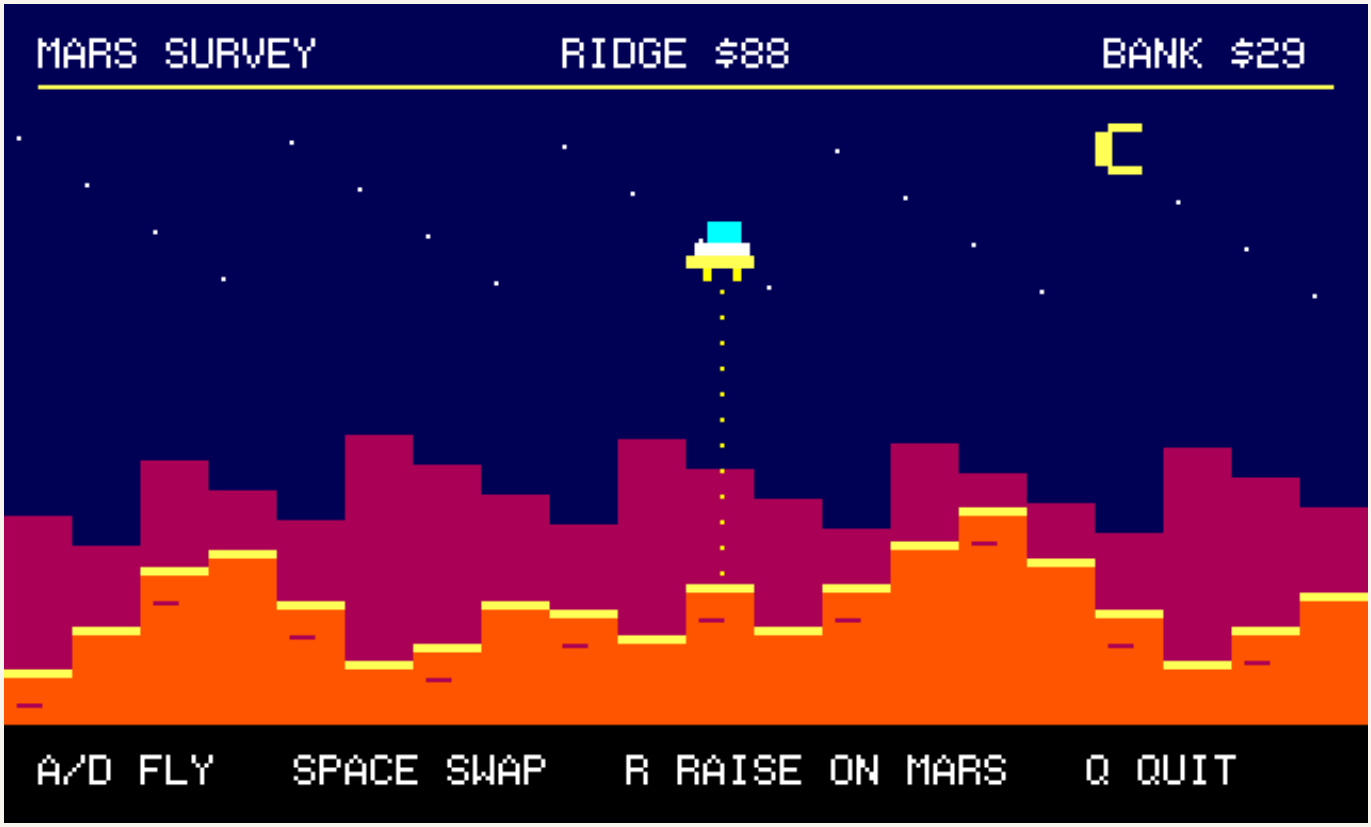


Figure 27: The running program on Mars after raising the ridge beneath the probe and visiting Earth.

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.

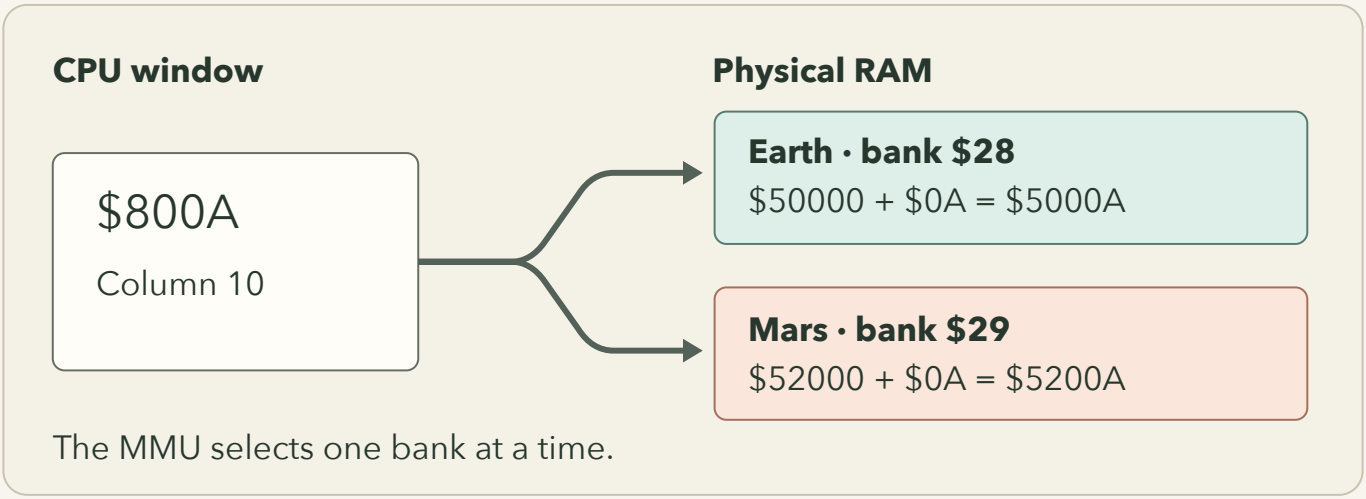


Figure 28: The same CPU address, $800A, reaches Earth at physical $5000A while bank $28 is selected, or Mars at $5200A while bank $29 is selected.

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.

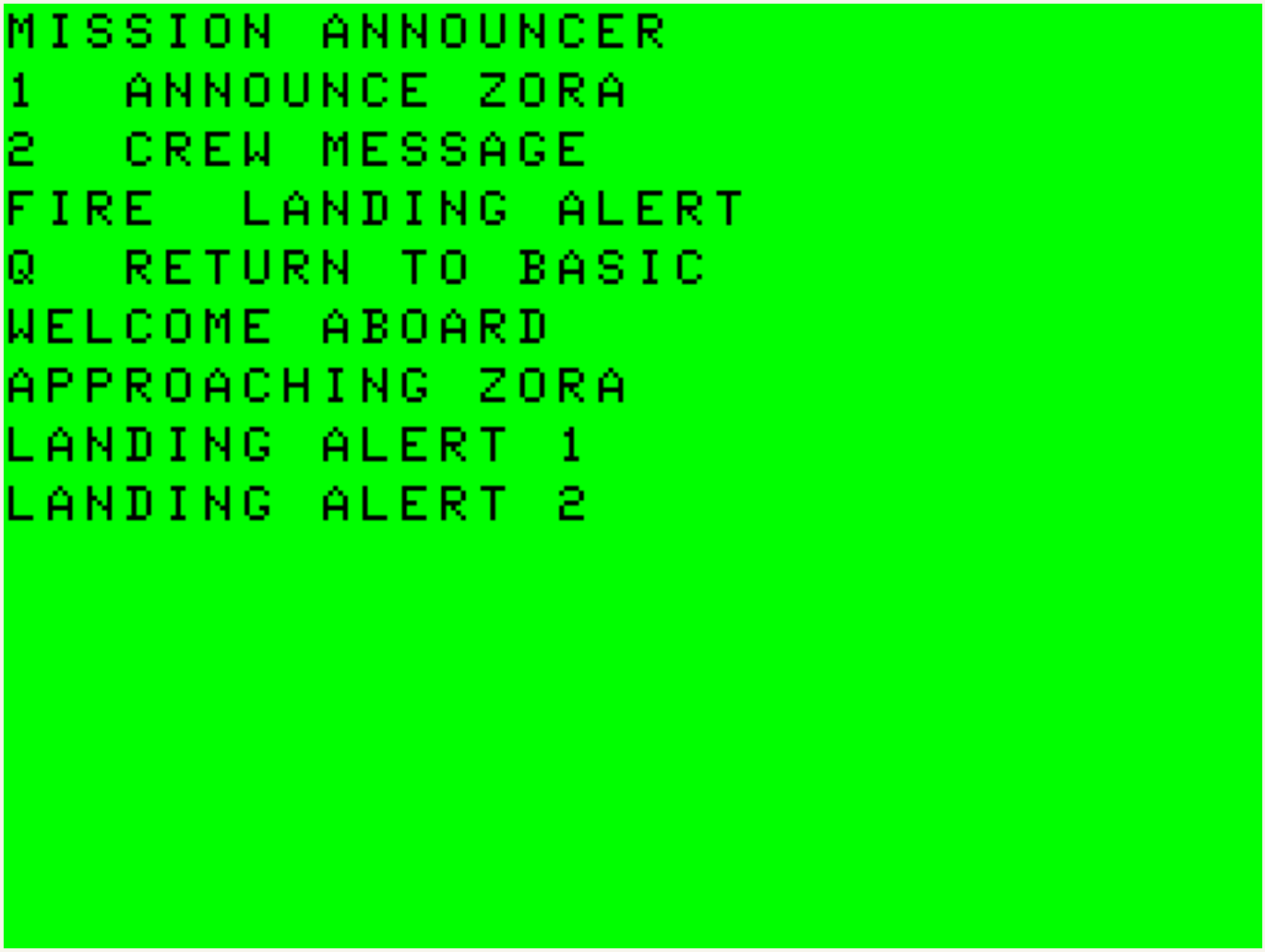


Figure 29: Mission Announcer shows APPROACHING ZORA and two counted landing alerts on the guest screen.

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.

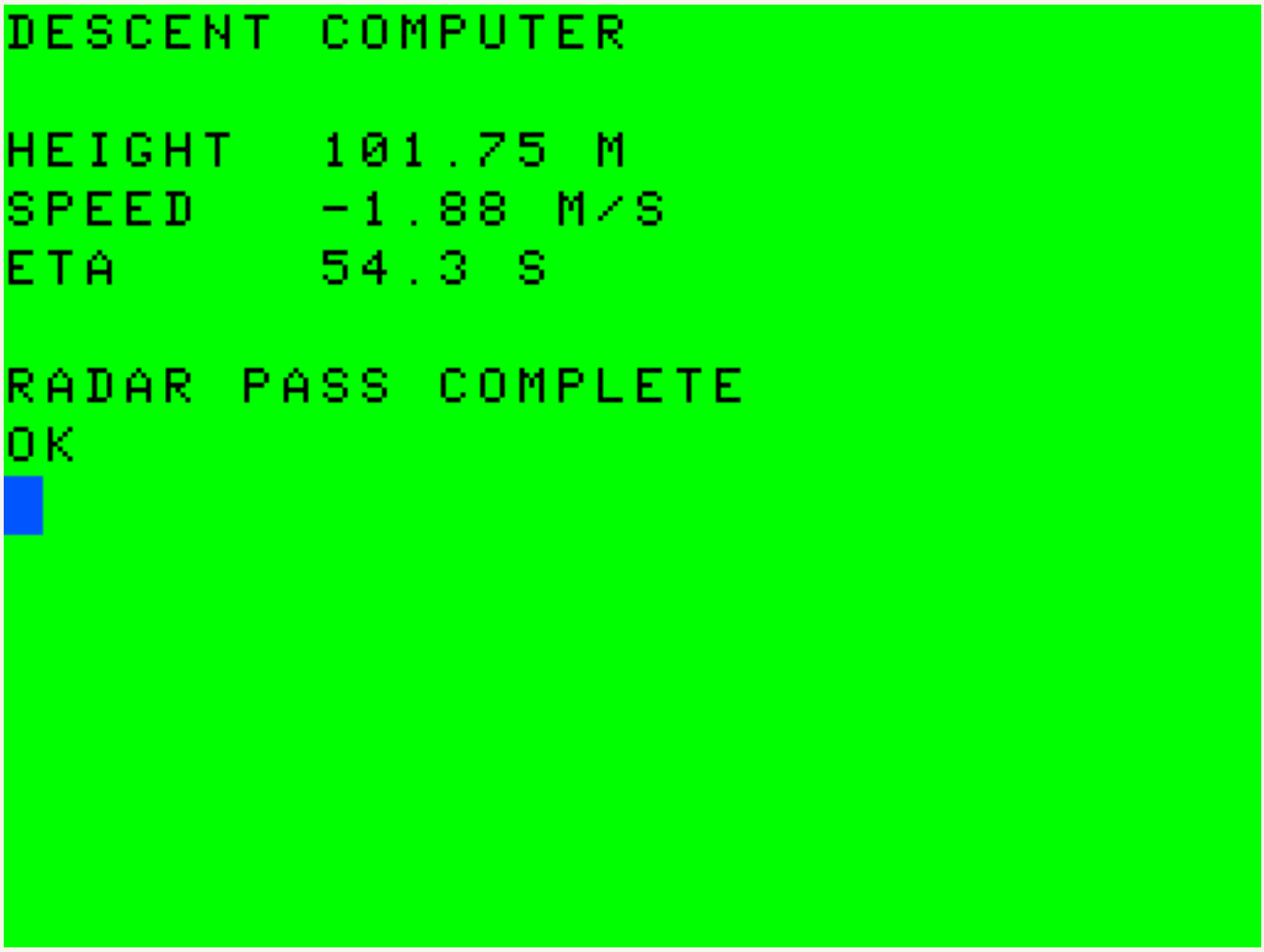


Figure 31: The Descent Computer guest display reads HEIGHT 101.75 M, SPEED −1.88 M/S and ETA 54.3 S.

```
    /* 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.

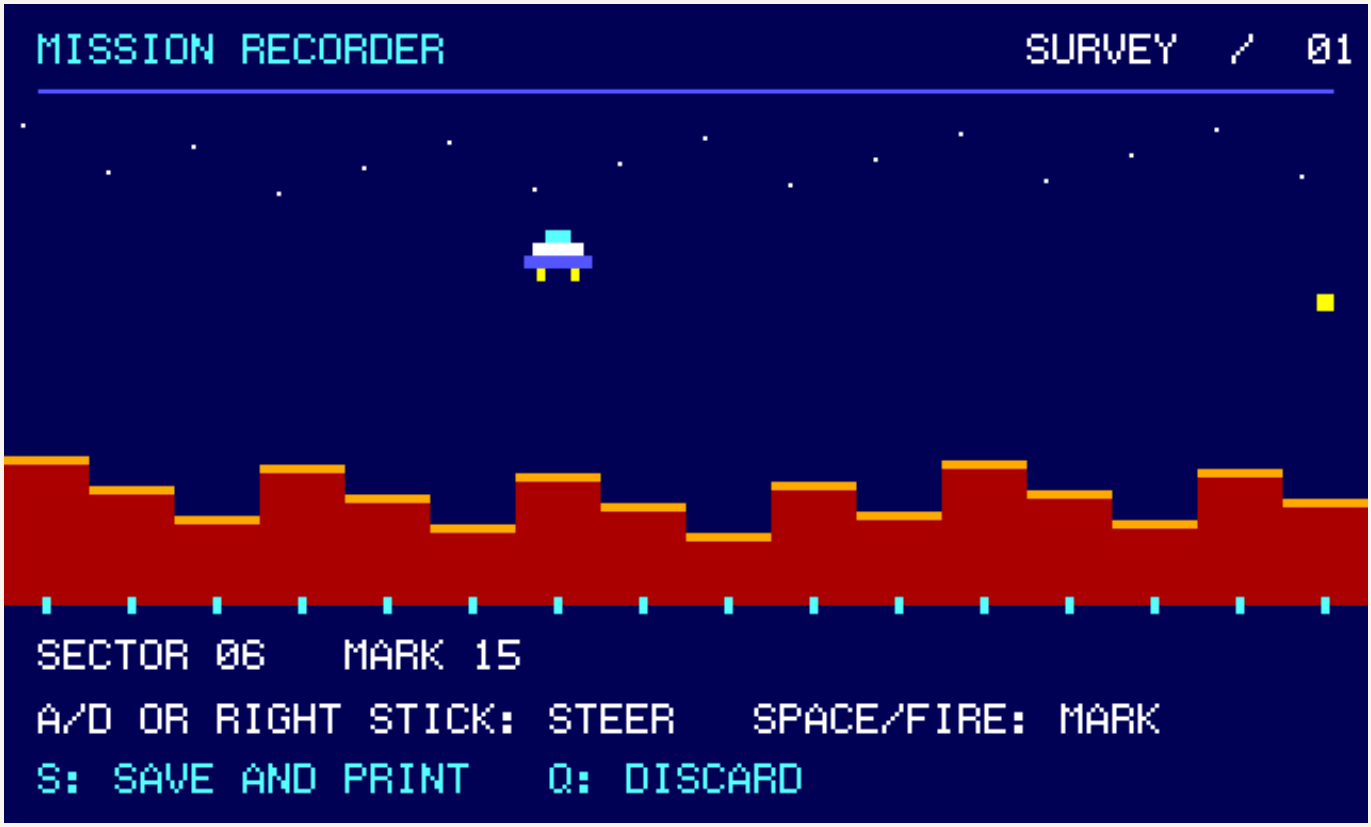


Figure 33: The flight display reads SECTOR 06 and MARK 15. The gold marker remains at the far right, where the pilot recorded it.

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.

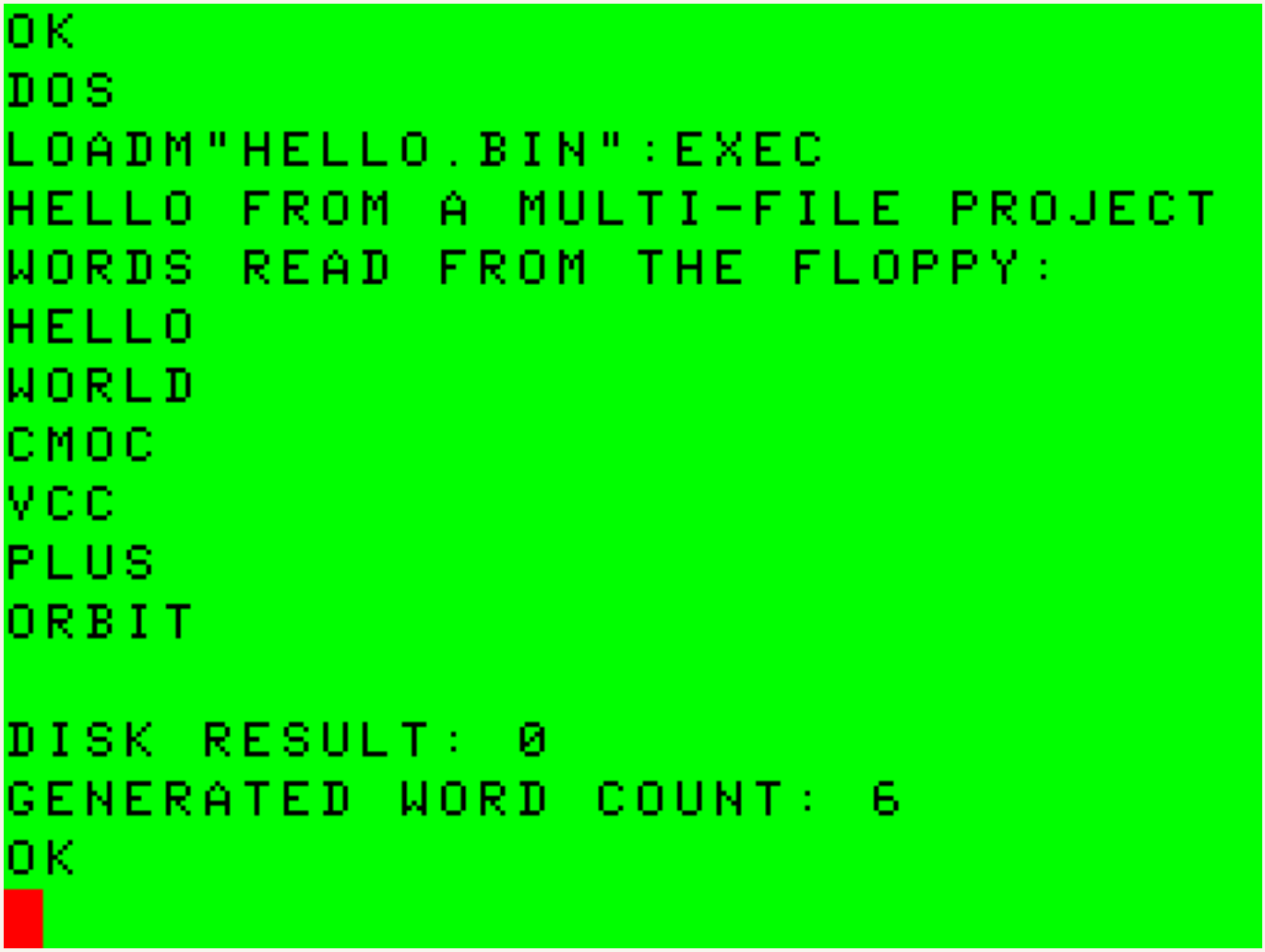


Figure 34: A fresh Disk BASIC guest ran the copied disk with DOS. ORBIT appears after PLUS, and the generated count is now 6.

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