ROM OS-9 Into Your CoCo

Boisy G. Pitre

Imagine turning on your CoCo and instead of being greeted with the all-to-familiar Microsoft BASIC copyright message and OK prompt, you see "OS9 BOOT". In two seconds, the OS-9 copyright banner and an OS9: prompt appears, and you have an OS-9 shell at your fingertips without having anything plugged in the CoCo's expansion port! It may sound far out, but it's actually quite easy to set up a ROM-based OS-9 CoCo. It can be done simply by replacing the Microsoft BASIC ROM that comes in all CoCo systems with a ROM containing OS-9 with special modifications to certain modules to allow booting from ROM.

OS-9 Sans Disk?

It may sound strange that OS-9 could be run totally from ROM, but indeed, this is the way that OS-9 was designed to run: without any disk storage. In the embedded OS-9 world, a disk drive or other mass storage device is incidental, or in most cases, impractical. Yet in the CoCo world, we have become accustomed to thinking of OS-9 as a Disk Operating System (DOS) as opposed to an embedded operating system. This was reinforced by the fact that Tandy/Radio Shack catalogs stated that OS-9 required a disk drive, and indeed, for the particular boot strategy that Tandy and Microware employed, a disk drive was and still is necessary.

So if it is possible to use OS-9 without a disk drive, then what's the point? How useful can OS-9 be without a disk drive? The answer: extremely useful. Since OS-9 provides multi-user and multi-tasking services along with a broad range of system calls, it makes an ideal embedded operating system for certain type of monitoring and controlling applications. Add to that the CoCo's I/O on-board I/O devices which can be interfaced to the real-world, and you have an ideal embedded hardware/software combination. Plus, the plethora of old used CoCos that can be found in thrift stores and flea markets insures that there are plenty of potential systems out there (I recently picked up a CoCo 2 at Goodwill for only $10 with joysticks and a cartridge game).

The Inspiration and the Challenge

I was inspired to embark upon this project after returning from the Chicago CoCo Fest in April 1998. At that fest, I watched as Color Computers were practically being given away at booths and at the Glenside auction, and I began to wonder how those systems could be salvaged from the closet and be put to use. The challenge was to make use of the basic CoCo unit with as little hardware hacking as possible and little or no use of external peripherals in order to minimize overall cost and space usage.

It was obvious to me that the limiting factor of the basic Color Computer was the BASIC ROM. Without a disk drive present to load programs, all one would ever get on power-up would be a multi-colored cursor. So replacing the BASIC interpreter with something more useful was an obvious start. The immediate solution was OS-9, since it was available for the CoCo and widely used. After some time was spent disassembling various OS-9 system modules and studying the BASIC ROM code for the CoCo 2 and CoCo 3, I devised a scheme that would allow OS-9 to boot totally from ROM.

For this project, I used two different CoCo 2 systems. One was a 26-3127B (64K Extended Color BASIC) and a 26-3134A (16K Standard Color BASIC). Obviously, I upgraded the later CoCo 2 to 64K to accommodate OS-9 Level One. The 26-3127B CoCo 2 contained a 16K ROM which was soldered to the motherboard, so it was necessary for me to take a soldering iron, remove the ROM and place a 28 pin socket in its place. The 26-3134A CoCo 2 already had a 28 pin socket installed, since Tandy intended for the 16K Standard Color BASIC machine to be upgraded at some point to Extended BASIC, so no soldering on that CoCo 2 was necessary.

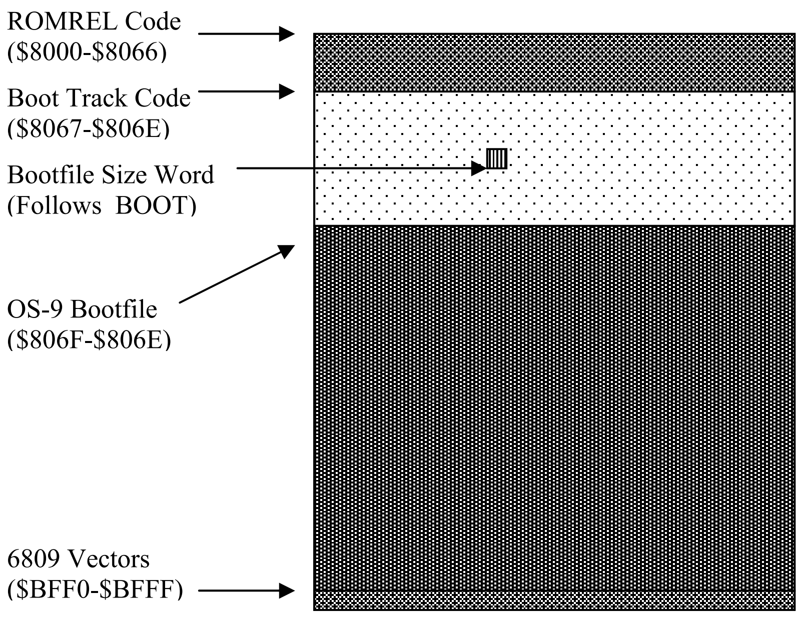
Examining the CoCo 2 Power-up Process

Before we can proceed with our project, we need to learn what happens when the CoCo 2 powers up. On power up, the CoCo 2's 6809 processor JMPs to a two-byte address that is held in the vector $FFFE. The contents of $FFFE and $FFFF can be seen from within BASIC by typing:

```
PRINT PEEK(&HFFFE)*256+PEEK(&HFFFF)
```

On my CoCo 2 system, the value returned is 40999, which is the address of the first instruction executed when the CoCo 2 is powered up. Is address 40999 always the address of the first instruction? That depends on what the CoCo 2's ROM contains. The word that is placed into address &HFFFE can also be

Figure 1 – OS-9 ROM Layout



found at address &HBFFE, which is the last 2 bytes of the Color BASIC ROM (The ROM starts at $8000 for Extended BASIC machines, or $A000 for Color BASIC machines). In fact, the last 16 bytes of ROM starting at &HBFF0 are mapped to the address &HFFF0. This is because the 6809 processor uses addresses &HFFF0-&HFFFF as vectors for various types of interrupts. If we changed any of the last 16 bytes of the BASIC ROM, then burned and installed the modified ROM, we would essentially change the contents of the 6809 vectors the next time the CoCo 2 was powered up. Using this knowledge, we can change the address where the 6809 starts executing in our OS-9 ROM.

The CoCo 2 offers us a 28 pin ROM socket that can accommodate a 27C128 EPROM, which holds 16K of code. Assuming our entire OS-9 ROM can fit within that 16K (minus 16 bytes for the 6809 vectors we just discussed), then we are on our way to a successful completion of the project's goals.

Examining the OS-9 Boot Process

A second prerequisite in completing the project is to understand how the process of booting into OS-9 from BASIC works. The most common way to boot into OS-9 from BASIC is through the DOS command. How does the DOS command work? It seeks to track 34 of the floppy disk and copies the 18 256 byte sectors into RAM location $2600 and JMPs to $2602. The code there sets up the CoCo's screen and memory then jumps into a module called `OS9`, which brings up the rest of the operating system. In order to get OS-9 to work from ROM, there must be some method devised to emulate what the DOS command does.

Furthermore, once control is passed to the `OS9` module, it calls into another module called `BOOT`. The `BOOT` module as supplied by Tandy/Microware attempts to locate the OS-9 bootfile on floppy disk. Since we are booting completely from ROM, we will be required to make changes to the `BOOT` module to find the OS-9 bootfile that is already stored in the ROM.

The Approach

Since we know that we can direct the 6809 to start executing code from any location in memory by changing the last two bytes of the ROM, we can create a ROM footprint that contains the necessary OS-9 code. We also need some special startup code that will copy the contents of the ROM to $2600 in RAM and JMP to $2602 just like the DOS command would. This startup code, located in a system module called `ROMInfo`, would also need to set up the CoCo's hardware to a known, stable state so that interrupts can work properly, PIAs are programmed, etc.

Writing the code for `ROMInfo` (see Listing 1) proved to be the most difficult part of the project, partly due to the fact that I had to study the disassembled BASIC code that was executed each time the CoCo was turned on. The Color BASIC code provided by Tandy does more that just set up the BASIC interpreter. It has to touch areas of memory that control the operation of the CoCo itself; thus, all of the relevant code had to be duplicated in the OS-9 ROM.

Figure 1 shows the organization of the 16K OS-9 ROM as it appears in the CoCo 2's memory space. The `ROMInfo` module is located at $8000, or at $0000 into the actual ROM. Following ROMREL is the rest of the boot modules. At $2E00 is the boot track code ($11F0 bytes).. Finally, the last 16 bytes of the ROM contain vector addresses that are used by the 6809. The last two bytes ($BFFE-$BFFF) contains the address of the first instruction to be executed by the 6809 when powered on. Since the first instruction is at $8015 (the entry point into the `ROMInfo` module), then the word at $BFFE is $8015.

`ROMInfo`'s job is to get the CoCo hardware properly setup. It then copies boot track at $2E00 to lower memory at $2600 and copies the Boot file at $8000 to $3800, and finally JMPs to $2602, just like the DOS command does. At this point, the Tandy/Microware supplied boot track code takes effect and starts the process of booting OS-9 Level One.

As mentioned earlier, the `BOOT` module needs to be modified to look for the bootfile in memory instead of on disk. Listing 2 shows boot_rom.a, which allocates memory for the bootfile, then copies the bootfile copied from ROM earlier by `ROMInfo`. The offset to the OS-9 bootfile in RAM is at $2600+$1200 ($3800), which immediately follows the boot track code. We are safe in copying from this area since the memory area allocated by F$SRqMem in the `BOOT` module and the area at $3800 don't overlap. However, since the OS-9 bootfile can be of varied length, the `BOOT` module needs to know just how many bytes allocate for the bootfile and to copy from $3800 over to the newly allocated area. This is accomplished by reserving two bytes immediately after the `BOOT` module for the size of the bootfile. Since the word is stored outside of the `BOOT` module, there is no need to validate any module CRC when the bootfile size changes; simply use a binary editor to modify the bytes. Once `BOOT` has allocated the memory for the bootfile and copied it from low RAM into the allocated area, it returns control to the `OS9` module.

Choosing the Bootfile Components

It is important to keep in mind that the CoCo gives us only 16K of ROM to work with. This means OS-9 Level One and our application must all fit in that 16K. Referring back to Figure 1, we see that the bootfile takes up $2E00 bytes. Although this still gives us more than half of the total ROM space, it is still prudent to choose our modules carefully.

The Tandy/Microware provided OS-9 Level One Vr. 2.00 bootfile contains the modules show in Table 1 and whether they are needed in a ROM based system or not. The first four modules listed, `OS9`, `OS9P2`, `INIT` and `BOOT`, are actually located in the boot track portion of the ROM, and thus will always be required; hence, their sizes aren't listed. The remaining modules however, are loaded later from the bootfile and their sizes are relevant for analyzing.

Table 1 – Modules in Tandy/Microware OS-9 Level One bootfile

Module	Function	Size in Bytes	Required
`os9`	Part 1 of OS-9 kernel	Boot track	Y
`os9p2`	Part 2 of OS-9 kernel	Boot track	Y
`init`	Kernel initialization module	Boot track	Y
`boot`	Booter code	Boot track	Y
`ioman`	I/O manager	1811	Y
`rbf, ccdisk, d0, d1`	Disk modules	4433	N
`scf,ccio,co32,term`	Serial/VDG modules	4083	Y
`printer,p`	Printer modules	467	N
`rs232,t1`	Serial port modules	453	N
`pipeman, piper, pipe`	Pipe modules	583	N
`clock`	Clock module	379	Y
`sysgo`	System initial process	363	Y
`shell`	Command line interface	1274	Y

Remember, the area in ROM for our bootfile to fit in is $2E00/11,776 bytes. If our ROM based bootfile contained the above modules, we would have a total bootfile length of 13,846 bytes, which exceeds the 11,776 byte amount. Obviously, we need to see what modules can be eliminated.

Right away, we can remove `rbf`, `ccdisk`, `d0` and `d1` since they are disk modules and aren't needed in a ROM based system. This removes 4,433 bytes from the bootfile, giving us a bootfile of only 13,846-4,433, or 9413 bytes. Also to reclaim even ROM space for our application, we can eliminate `pipeman`, `piper` and `pipe`, assuming we don't need pipes in our application. Removing the pipe modules leaves us with a bootfile size of 9,413-583, or 8,830 bytes. The same goes for `printer` and `p` if our application doesn't use the printer; the bootfile size is now reduced to 8,363 bytes. Although `rs232` and `t1` are marked as not required, they may come in handy for introducing new modules into the computer through a combination of a system module and utility that we'll talk about later. For now, we'll leave them out, shrinking the bootfile even further to 7,910 bytes. `ROMInfo`, whose size is currently 143 bytes, must be added to the size of the bootfile, bringing the size of the bootfile up to 8,053 bytes.

The `clock` module is a must, since it gives OS-9 the ability to multitask and also keeps track of the time. `Sysgo` is needed to set up initial I/O paths and fork the initial process, which is the next module, `shell`. `Shell` is not necessarily required (your custom application could be the initial process in your own embedded application) but for purposes of the current discussion, we'll leave it in for now. With a bootfile size of 8,053 bytes and a total maximum bootfile size of 11,776 bytes, that gives us 11,776-8,053, or 3,723 bytes for our application. That's going to be way too small for `RunB` and a packed Basic09 program (even the total ROM size, 16K, is pushing it for just the run-time module and a program alone!). The leftover size is also inadequate for all but the simplest C program, but it is plenty of room if we can compromise and write our application in assembly language. Table 2 shows the modules in our new ROM bootfile.

Table 2 – Modules for a bare-bones ROM boot

Module	Size in Bytes	Required
`rominfo`	143	Y
`ioman`	1,811	Y
`scf,ccio,co32,term`	4,083	Y
`clock`	379	Y
`sysgo`	363	Y

`shell`	1,274	Y

Limitations of Running from ROM

An obvious consequence of running out of ROM is that you are seemingly limited to the modules that you have made available in the ROM; there seems to be no apparent way to introduce new modules into the ROMmed CoCo. In most embedded system designs this won't be a severe limitation since the computer is programmed with only a certain, unchanging set of tasks to perform.

However, there may be times when it is advantageous to introduce new modules into the OS-9 system after it has been booted from ROM. Perhaps there are other modules that are required to be in memory, but due to limited ROM space, they were not able to be fit on the ROM. Uploading these modules after boot-up may be an acceptable solution. However, there are two problems to overcome: (1) how do we bring the modules into the system and (2) once the modules have been brought into the system, how do we tell OS-9 about them?

The answer to our first problem is an already existing piece of hardware on board the CoCo – the serial port. By including the `rs232` and `t1` modules, we open a gateway by which data can pass into and out of our CoCo to other computer systems through their serial port with the construction of a null modem cable. The only limitation of the CoCo's built-in serial port is that it is reliable only at 300 baud, which is extremely slow for downloading large modules. If you have an RS-232 pak available that plugs into the expansion port, you can include the `aciapak` and `t2` modules as part of your boot ROM instead of `rs232` and `t1`, and download modules at much higher baud rates. Once the serial port has been established, then it is a simple matter of writing software to open a path to the port and read in data (in this case, an OS-9 module) into an allocated portion of RAM.

Our second problem, alerting OS-9 to the presence of downloaded modules, involves a bit more work and creativity. Let's say that our theoretical program allocates the memory and reads the module from the serial port into that allocated memory area. OS-9 must be told of the existence of the new module in such a way that it can insert a reference to the new module into the module directory. Looking through the OS-9 Technical Reference, we see that there is such a system call, F$VModul. This system call is passed a pointer to a data area containing a potential OS-9 module. OS-9 then validates the module, and if CRC checking confirms that the module is intact, it will be introduced into the module directory.

It seems that the F$VModul system call will take care of our problem; however, there is a slight snag. The F$VModul system call is a privileged mode system call. That is, it must be called from system state. Our program that allocates the memory and reads in the module would be running in user state, and therefore would be ineligible to use the F$VModul system call. It would seem that the answer to our problem has revealed yet another problem that we must overcome.