The Loader Problem

By William Astle
December 9, 2011

No matter what you're writing, any nontrivial software for the Coco3 needs some sort of scheme for loading it. The scheme is trivial for a BASIC program or a simple machine language one. But what if you have multiple chunks of data, say level graphics in a game, that need to be loaded all over memory. Or maybe you want people to be able to LOADM your program and not have to do anything else. Or maybe you simple have a program that is too big to fit in the usual memory space without conflicting with BASIC. Or maybe you are going to take over the bare metal and you need a way to load your program over top of Basic. Whatever your reason, there is a solution.

There are several ways to handle the loader problems. Each has advantages and disadvantages. I will examine several possibilities below and discuss the merits and drawbacks each one has. I will generally progress from simpler to more difficult. All assume a stock Coco3 except where otherwise noted. Also, all assume you will be loading some sort of machine language program.

Annoying Book Keeping

Example listings are provided only for the particularly complex schemes. You are free to use any example code provided below in any way you see fit but with the caveat that if it fails in any way, it is your sole responsibility.

Additionally, while I believe this document to be accurate, it is nearly inevitable that there are errors. There may be updated versions of this document available from time to time. The canonical location for it is always in the Coco section of http://lost.l-w.ca/. However, permission is granted to copy, distribute, and otherwise mangle it as you see fit so long as no attempt is made to attribute your (or other people's) modifications to me.

BASIC Loader

The simplest method of loading your program is to write a short BASIC program. This method works best on a disk system but it can work also with a tape system. In this instance, the BASIC program can put up progress messages on the screen as it loads each chunk of the machine language program and does any book keeping between loads. So it can switch around MMU blocks, load a chunk, switch MMU blocks again, load another chunk, and so on.

The biggest advantage of this method is that it is simple and it is very easy to adjust the loader process for additional data chunks or to do additional setup as needed. The primary disadvantage of this method is that it requires multiple files. On a tape system, this is not necessarily a problem. However, on a disk system, due to the file system format used by BASIC this may prove to be inefficient and may artificially limit how much you can put on a single disk.

Additionally, this method requires the loader know whether you are using tape of disk to load the remainder of the program. This is true of most loaders and is not a major handicap since when packaging for a particular distribution medium, the appropriate loader can be used.

Multi-Origin Binaries

The next simplest loader method is using multi-origin binaries. This might actually be simpler than the BASIC program above but it is actually quite a bit less flexible.

Multi-origin binaries exploit the format LOADM and CLOADM expect for binary files. It should be noted that this method will work on all versions of the Coco as long as Extended BASIC or higher is installed. Plain Color BASIC does not support the multi-origin binary format.

A description of this file format is helpful before proceeding further. Each file consists of a series of blocks of various lengths. Each block begins with a five byte header followed by any data for that block. There are two header formats. The first is the preamble. A preamble has its first byte set to zero. The second and third bytes specify the length of the data in the block and the fourth and fifth specify the address in RAM to load that block of data to. Then the block of data to be loaded follows the preamble immediately. The other type of header is a postamble. A postamble has its first byte set to FFh. The second and third bytes of a postamble are ignored and the final two bytes specify the execution address of the program just loaded. The postamble also signifies that loading the file is complete.

Multi-origin files exploit the file structure described above to cause code and data to be loaded at convenient locations all over memory. Thus, it is possible to load a chunk into the memory used by the 32 column text screen, load another chunk at 7000h, load another message over the text screen, load a chunk at 4000h, and so on.

This method has one severe limitation, however. Every time LOADM or CLOADM writes a byte to its destination in memory, it reads it back again to check if it was written correctly. While this is not a problem for any memory location that specifies actual RAM, it does present a problem for many addresses in the I/O space. That is because many of those addresses are write-only or are incomplete. For instance, when reading the value of an MMU register, the upper two bits contain random junk. While that random junk is somewhat predictable, it cannot be relied upon to be the same between real hardware and various emulators. Thus, preloaded the junk bits with what they will probably be when the verification step is done is risky. With the MMU, this risk is further compounded by the fact that there exist 1MB and 2MB memory boards that actually use those bits even if they still are not readable.

The practical upshot of the preceding paragraph is that this method cannot be reliably used to load data to blocks of memory outside the usual 64K memory map used by BASIC. It does, however, provide the groundwork form some more elegant solutions.

Machine Language Loader

Suppose, instead of putting your entire program into a single multi-origin binary file, you split it up into chunks. The first chunk loaded is simply a program that knows how to load the remaining chunks. This is very much similar to the BASIC program loader method described above except the loader program is machine code instead.

This method has a few gotchas, however. If you are using a tape system, the ROM calls to handle reading each subsequent file from the tape are simple enough and their addresses can be relied upon. Additionally, the tape format is constant. Unfortunately, the same does not hold true for disk systems.

If you will be relying on the disk file system for this method, the most reliable means of reading additional files is to implement the file system code yourself. That is, you must read the directory track, follow the granule chain, and read the various sectors of the file into memory yourself. This is not terribly difficult code but it is another possible source of errors. However, as long as you restrict

yourself to calling DSKCON via the officially sanctioned vector at C004h, this method is pretty much guaranteed to work with any Disk BASIC ROM, even those extended for such products as Drivewire or hard drive interfaces.

If you do not want to implement the file system code yourself, you can also cause BASIC to interpret various commands directly, but this is even more confusing to make work. It will also vector back to the BASIC error handling code in the event of an error.

This method can, however, do anything it wants in the loader program, including completely ignoring BASIC and doing its own thing. The only restriction is that the initial loader program must be loadable and executable within BASIC.

This method has all the drawbacks of using a BASIC loader as well as the additional complexity of writing the loader itself in assembly language. This method can work on any Coco.

Autoexecuting Loader

Have you ever seen those programs that start running as soon as you LOADM or CLOADM them with no need to enter EXEC? Have you ever wondered how to do that? The answer is simple and complicated at the same time. They all exploit our old friend, the multi-origin binary file.

Because the standard binary (or LOADM) format can place chunks of bytes at any location in memory, it can actually be used to overwrite variables BASIC uses for book keeping. Autoexecuting programs exploit this fact in various ways to intercept BASIC before it returns control to the user. While all cases use multi-origin files, exactly how they use them varies wildly.

Interrupt Hooks

In this case, the last block loaded from the file overwrites the IRQ vector at 10Ch. Then, as soon as the next IRQ occurs, control transfers to an address specified by the loaded binary instead of to the usual interrupt handler. This method will work with any Coco but it has a couple of disadvantages. The first is that it is possible for the interrupt to occur while the vector is being overwritten. If that happens, the entire machine could crash. The other is that if your program wishes to coexist with BAISC, it will have no idea what value to put back into the IRQ vector since that vector contains different values depending on the presence and version of Disk BASIC. If you intend to take full control of the hardware, however, this method is potentially viable. It will, however, leave the tape or disk motor running.

Trapping the Main Loop

Another method takes advantage of the fact that every time through the command interpretation loop, BASIC does a "JSR 019Ah". This is otherwise known as a ramhook. This loop will be executed at least once upon completion of LOADM or CLOADM. Thus, we can overwrite the three bytes at 19Ah with a JMP to our own code. This will work almost exactly like the interrupt hook method except it does not suffer from race conditions. It does, however, suffer from the problem of not knowing what was already in the vector before it was overwritten. If, however, you intended to take full control of the hardware, this is not a problem. This has the further advantage that the file being read will have been properly closed and the IRQ that handles turning off the disk motor will still be active. This method will work on any Coco.

Trapping Close

Another vector that can be useful to trap for this purpose is the close vector at 0176h. It works the same as trapping the main loop except the file has not been closed so the tape or disk motor may continue running. It also suffers from the same problem as the other vector intercepts. That is, you don't know what was there before you overwrote it. Again, if you intend to take full control of the machine, this is not a problem.

Direct Patching

This method takes advantage of the fact that the Coco3 is running in RAM all the time. That means that you can actually overwrite parts of BASIC using LOADM without encountering the verify problem described above. However, it can only be relied on to work on the Coco3. Even then, it can only be relied on when overwriting portions of the Color BASIC, Extended BASIC, or Super Extended BASIC ROMs. It cannot be used reliably for overwriting portions of Disk BASIC. In this case, trapping the main loop can be done by overwriting the JSR 19Ah at AD9Eh and trapping the close call can be done by overwriting the JSR 176h at A42Dh. This has the advantage that you know precisely what was overwritten and you can restore it properly if you need to coexist with BASIC. Of course, other things can be patched directly for other purposes, too.

Single File Loading

You may wonder why I mentioned trapping the close call above since trapping the main loop is clearly equally good. Well, trapping the close call allows for a nifty trick where the data you need to load is present in the same file you already loaded the loader program from. I can hear you all saying, "Load my what from where now?" but bear with me while I explain.

This method exploits the way LOADM (and CLOADM) are implemented. It is implemented as follows:

1. Open the specified file
2. Read 5 bytes (the block header)
3. If the first byte is FFh, treat it as a postamble and go to step 5
4. Read the specified number of bytes, one by one, from the file and place them in the specified memory location. When complete, go back to step 2.
5. Set the default EXEC address.
6. Close the file.
7. Return to the BASIC main loop.

Note that in a properly formatted file, in no case does the EOF indicator serve any purpose. As soon as the postamble is read, the file is closed. It is this fact that can be exploited for single file loading. Immediately after the postamble, we can place anything we want and it will not interfere with the LOADM process, nor will it be loaded anywhere without our intervention.

The question now becomes how do we read that data given that we do not know what the name of the file is, whether it is on tape or disk, or the precise entry point for the relevant "open file" routine. This is where trapping the close call comes in.

Once we install our trap on the close routine, when our replacement executes, we know the postamble has been read. We also know that the file is still open and is positioned in the correct place within the

file. Now, all we need to do is read the remaining bytes in the file, until EOF or other condition we specify, and do what we will with them. These bytes can be in any format we desire, including a raw blob that our loader program somehow knows how to organize.

Either method of trapping the close call will work but the direct patch method is preferred if you intend to continue to coexist with BASIC. If, however, you will be operating on a Coco2 or Coco1, you must use the ramhook vector to trap it. If you will continue to coexist with BASIC, you must restore the original values at the location you overwrote to trap the close call. Thus, the direct patch method is easier because you know precisely what you overwrote.

Single File Example

Because this is probably all Swahili mixed with Klingon to most of you, the following is an example of the broad structure of how the source might look. In this example, the "payload", the bytes that follow the postamble, is a series of chunks that specify an MMU block number (1 byte), a length (2 bytes, not greater than 2000h), and the block data. EOF indicates the end of the data.

```
            ORG $2000
LOADER      LDD #$176         ; restore close code
            STD $A42E
NEXT        BSR GETBYTE       ; fetch byte
            BCS EOF           ; if EOF, we're done
            STB $FFA2         ; set MMU block
            BSR GETBYTE       ; fetch MSB of length
            BCS EOF
            TFR B,A
            BSR GETBYTE       ; fetch LSB of length
            BCS EOF
            TFR D,Y           ; put length where we can use it
            LDX #$4000        ; point to memory block
LOOP        BSR GETBYTE       ; fetch byte
            BCS EOF
            STB ,X+           ; store it
            LEAY -1,Y
            BNE LOOP          ; keep looping if not done
            BRA NEXT          ; look for another block
EOF         JSR $A42D         ; close the file
            LDA #$3A          ; restore MMU
            STA $FFA2
            JMP RUNPROG       ; run the main program code
; this routine fetches a byte from the file and places it in B
; if EOF is reached, returns with carry set, otherwise clear
; NOTE: COM sets carry always and CLR clears carry always
GETBYTE     PSHS A
            JSR $A176
            TST <$70
            BEQ NOTEOF
            COMA
            PULS A,PC
NOTEOF      CLRB
            TFR A,B
            PULS A,PC
; your program code goes here
RUNPROG

; this final bit installs the close trap
```

```
; the "END LOADER" line sets the exec address to LOADER
; but is not strictly necessary because this code autoexecutes
            ORG $A42E
            FDB LOADER
            END LOADER
```

Astute readers will notice that this code only ever calls the Color Basic ROM routines. It is that fact that allows it to work for tape files, disk files, or even some funky scheme that someone has patched into the BASIC ROM (say Drivewire or hard drives or what have you). It uses only two routines, "Console In" and "Close".

I don't have to point out that you can do anything you want in the LOADER code. You do not have to restrict yourself to loading memory blocks like the above. What is possible is limited only by your imagination. Also, in the above case, the bit that says "your program code goes here" can contain your entire program if it fits in the usual BASIC memory map.

This example combines the autoexecuting trick with loading extra data from the end of the file. In this particular case, that extra data might be a high resolution graphics image or sprite data, or even auxiliary code that the main program knows how to call later. Thus you could have a program with many dozens of kilobytes of actual code and data load from a single file.

This example still has one problem in the case where you wish to take full control of the machine. You cannot load anything over top of BASIC or its variables and buffers while you are loading.

Two Stage Loading

Now that you have fully grokked the single file autoexecuting scheme above, you have decided that you no longer need BASIC once you have loaded your program and you really want to gain that 39.5K of memory that BASIC prevents you from using, and you want to load your code into high memory where BASIC lives. You cannot do that using any of the methods above because as soon as you start overwriting BASIC, the loader will crash because you are overwriting routines it relies on. Fear not, however, for there is a solution: two stage loading.

By using two stages, it is possible to use BASIC to read the entire program from the disk or tape file but still have it overwrite portions of the BASIC code during the loading process. This method does require more memory because it requires a temporary memory space exactly the size of the payload that must be loaded and interpreted in addition to where the payload will finally live. There are tricks to reduce this overhead but they are beyond the scope of this document.

Basically, a two stage loader is exactly the same as a one stage loader as described above. The difference is that the payload contains a the program which interprets it and the first stage loader simply reads the disk and then transfers control to the second stage loader. The second stage loader can be any program you care to write and can do anything needed to properly load your main program.

Two Stage Example

In this example, the first stage loader reads the payload into memory starting at physical address 0 and working upward. Once it has read the entire payload, it disables interrupts, puts physical memory block 0 into CPU block 0 and starts executing at address 0.

The second stage loader looks for its payload immediately following its code. It interprets the payload as a LOADM format string of bytes. This is convenient because all assemblers support generating this type of binary and it is flexible enough to do any sort of messing about with hardware registers and so on during the load process. It also has the advantage of keeping the loader code simple.

This method requires at least three source files. The first is the stage one loader which is assembled as a LOADM format binary. The second is the stage two loader which is assembled as a "raw" binary (headerless). The third is the actual main program which is assembled as a LOADM binary. The main program contains all data and code associated with it.

This example allows the main program to overwrite BASIC in high memory. It also works best on machines with 512K of memory.

Once all three portions are assembled, the final binary is created by appending all three files in the following order: stage one, stage two, main program. This is done byte for byte with no gaps in between.

Stage One

```
            ORG $2000
LOADER      LDD #$176         ; restore close routine to be polite
            STD $A42E
            CLRB              ; initialize block counter
LOADER1     STB $FFA2         ; set MMU
            LDX #$4000        ; init load pointer
LOADER2     JSR $A176         ; get byte from file
            TST <$70          ; EOF?
            BNE LOADER3       ; branch if so
            STA ,X+           ; save byte in memory
            CMPX #$6000       ; end of MMU block?
            BLO LOADER2       ; continue if not
            INCB              ; next MMU block
            BRA LOADER1       ; set MMU and continue loading
LOADER3     JSR $A42D         ; close file to be polite
            ORCC #$50         ; disable interrupts to prevent crash
            CLR $FFA0         ; set CPU memory block 0 to physical memory 0
            STA $FFD9         ; go to turbo mode
            JMP >0            ; transfer control to payload (stage two)
; this installs the close trap
            ORG $A42E
            FDB LOADER
; "LOADER" not really required here but is included for completeness
            END LOADER
```

Stage Two

There are a couple of notes about this source listing. It must be assembled in a headerless or raw mode. Also, the FDB instructions must not be replaced with RMB as that may result in incorrect output depending on the assembler.

Also, because only two bytes of stack space are required, only two bytes of space are allocated for the stack. If you change this code, you may need to expand the stack space. The stack is placed where it is to allow the payload as much access to the machine as possbile.

```
            ORG 0
LOADER      LDX #PAYLOAD+$2000 ; point to start of payload in logical block 1
            CLRA              ; initialize block counter
            STA >$FFA1        ; initialize MMU
            LDS #TEMPSTACK+2  ; temporary stack location
LOADER0     BSR GETBYTE       ; fetch block flag
            BNE LOADER2       ; branch if postamble
            BSR GETBYTE       ; fetch block length
```

```
            STB >TWORD
            BSR GETBYTE
            STB >TWORD+1
            LDY >TWORD        ; fetch block count
            BSR GETBYTE       ; fetch load address
            STB >TWORD
            BSR GETBYTE
            STB >TWORD+1
            LDU >TWORD        ; fetch destination address
LOADER1     BSR GETBYTE       ; read a byte to its destination
            STB ,U+
            LEAY -1,Y         ; finished block?
            BNE LOADER1       ; continue if not
            BRA LOADER0       ; handle next block
LOADER2     BSR GETBYTE       ; ignore two bytes
            BSR GETBYTE
            BSR GETBYTE       ; fetch execute address
            STB >TWORD
            BSR GETBYTE
            STB >TWORD+1
            JMP [>TWORD]      ; transfer control to main program
TWORD       FDB 0             ; scratch location
TEMPSTACK   FDB 0             ; temporary stack location
; read byte from A,X and bump pointer
GETBYTE     LDB ,x+           ; fetch byte
            CMPX #$4000       ; end of block?
            BLO GETBYTE0      ; return if not
            LDX #$2000        ; reset pointer to start of block
            INCA              ; bump block number
            STA >$FFA1        ; set MMU
GETBYTE0    TSTB              ; set flags on byte read
            RTS               ; return
PAYLOAD     EQU *             ; payload starts here
```

The Main Program

The listing for the main program is not provided since it depends on the precise main program. However, upon entry, the main program will have the following conditions. A and X will point to the first byte beyond the postamble with the MMU block number being in A and the block offset corrected for CPU block 1 in X. The MMU will still be set up exactly as it was during the stage two loader so a duplicate of the GETBYTE routine will continue reading bytes beyond the postamble. Thus it is possible to have the main program further interpret a blob of data following its code in the payload.

Also, upon entry, the machine will be in double speed mode and interrupts will be disabled.

The only restrictions on the payload is that it must not interfere with the MMU blocks in FFA0h and FFA1h. It also must arrange not to clobber the payload itself during the loading procedure. Otherwise, the payload can adjust MMU registers, memory locations, and so on as it sees fit.

Once the main program starts executing, the first things it should do are initialize any hardware as needed and set up a stack space.

Hybrid Loaders

While this is probably obvious to most, it is also possible to combine concepts from various bits above and create much more complex loaders. For instance, one could mix the one stage single file method

with the two stage method to good effect. The only limits to these methods are your imagination and the actual hardware limits of the Coco. Remember, there is nothing stopping you from having a BASIC program load several binaries, ask a bunch of questions, then turn control over to an autoexecuting loader that reads a bunch of stuff into memory then transfers control to a second stage which does further processing. What you do will depend on the precise requirements of your project.

Acknowledgements

I would like to thank user remz from the Games Workshop Forum for pointing out an error in the single file example above. I had originally written BNE in the GETBYTE routine where BEQ is, in fact, the correct branch instruction.