M. David Johnson
http://www.bds-soft.com
info@bds-soft.com

A Question of Unravelled Terminology

by M. David Johnson

2021/09/29

Abstract

A new reading of the Unravelled books at addresses $FFDE and $FFDF reveals either an error in the texts, or a potential for misunderstanding. In either case, it is confirmed that:

A POKE or other write into $FFDE sets RAMROM mode, and

A POKE or other write into $FFDF sets ALLRAM mode.

Table of Contents

Abstract 2

Introduction 4

General Methodology 5

Results 7

Conclusions and Future Work 8

Appendix A: Book Entries 9

Appendix B: My 1990 CF83 POLCAT Code 11

Appendix C: Prompt Attention 13

Appendix D: Changing the Language 17

Appendix E: CoCo Hardware Programming 18

Appendix F: SYSTST01 Code 19

Appendix G: SYSTST02 Code 23

Appendix H: Huffman & Castro Code 27

Appendix I: Clarity vs. Memory Space 31

Appendix J: My 64K CoCo 2 System 34

Appendix K: My CoCo Philosophy 35

Appendix L: New BDS Software License 37

Works Cited 38

Other References 39

Introduction

The four Unravelled books:

Color BASIC Unravelled II.
Extended Color BASIC Unravelled II.
Disk BASIC Unravelled II.
Super Extended BASIC Unraveled II.

as cleaned-up and re-issued by Walter Zydhek in 1999, have been an invaluable resource for those who wished to explore the inner workings of the Radio Shack Color Computer's system software; myself included.

I was thus momentarily stunned this past month, when I paused to actually look at a tiny piece of the books' code which I had always simply skimmed over because I thought I had always understood it perfectly.

That tiny piece of code (see Appendix A below for more details) is:

```
FFDE ROMCLR     RMB 1     ROM DISABLED
FFDF ROMSET     RMB 1     ROM ENABLED
```

which seems to me to be saying, that if you want to disable ROM and enter ALLRAM mode, you should POKE something into address $FFDE. Conversely, if you want to return to RAMROM mode, you should POKE something into address $FFDF

But I had always understood the exact opposite. To enter ALLRAM, you POKE something into $FFDF, and to return to RAMROM, you POKE something into $FFDE.

Now, it's possible that I'm misunderstanding the author's intent here but, if so, it's a misunderstanding which could bedevil newcomers to the CoCo as well. It probably wouldn't bother any of the old hands; they've probably been skimming over it for 20+ years just as I have.

But, in order to help new CoCoNuts avoid getting lost in these particular weeds, and because this promised a fascinating little investigation, I set out to explore the truth of the matter.

All the code presented in this paper is included in the **`UNRAVLD.DSK`** file which can be freely downloaded from http://www.bds.soft.com/cocoPapers.php . Please refer to Appendix L herein for a copy of the New BDS-Software License.

General Methodology

The first thing I did was to recheck the CF83 operating system I wrote back in 1990 (See Appendix B below). Since this OS has worked as intended for 30+ years, and since it switches from ALLRAM to RAMROM to process keyboard input, and since my POLCAT routine clearly wouldn't work at all if it didn't have access to the ROM routine, it seemed clear that my understanding of how to switch between the ALLRAM and RAMROM modes was correct.

I then explored the available literature on this matter. In the CoCo, the $FFDE and $FFDF addresses control the "TY" bit in the control register of the Synchronous Address Multiplexer (SAM).

The Motorola MC6883 SAM Datasheet was only partially helpful in this matter. Although the datasheet clearly delineates how the SAM does bank-switching between blocks of RAM and/or ROM, it is not CoCo-specific. The memory map type is controlled by the "TY" bit; bit 15 in the SAM Control Register. The datasheet indicates on page 8:

> This bit is soft-programmable (as are all 16 bits in the SAM CONTROL REGISTER,) and selects one of two memory maps. Memory Map #0 is intended to be used in systems that are primarily ROM based. Whereas, memory map #1 is intended for a primarily RAM based system with 64K contiguous RAM locations (minus 256 locations.)

But, again, which address lines control the "TY" bit is CoCo-specific and thus not addressed by the datasheet.

I did encounter articles and documents by Joseph Forgione (Appendix C), Marc Campbell (Appendix D), Chris Lamont (Appendix E) and Allen Huffman and Juan Castro (Appendix H), all of which concur with my understanding of how the CoCo's bank switching actually works.

Finally, I decided to test my understanding directly. Addresses $BFE6 through $BFEF are ten bytes, up near the higher end of the Color BASIC ROM, which *Color BASIC Unravelled II* describes as "Unused Garbage Bytes". I thus suspected that I could mess with these bytes without crashing the system.

I reasoned that if I could POKE a new value into one of these bytes, and if I could then read that same new value back via a PEEK, then I must be in ALLRAM mode.

Conversely, if I tried to POKE a new value into the byte and the byte remained unchanged, then I must be in RAMROM mode.

I thus wrote **SYSTST01.BAS** (See Appendix F) to conform to my understanding of what the Unravelled books were presenting; and **SYSTST02.BAS** (See Appendix G) to conform to what I believe is the correct method for switching between the ALLRAM and RAMROM modes.

I first ran the two programs under VCC 2.1.0d as it normally starts up, operating under Windows 10 Pro 64-bit. For both programs, I used the command **`RUN`**. The results were as displayed in Appendices F and G respectively.

Next, I ran the two programs under the same VCC configuration, but in CoCo 2 mode, i.e. I used the command sequence **`X=PEEK(&HFF90):X=X OR &H80:POKE&HFF90,X:RUN`**. The two sets of results were exactly the same as before.

Finally, I ran the two programs in my physical 64K CoCo 2 Setup (See Appendix J), after running the **`ROM2RAM.BAS`** program (described in Appendix H) to copy the ROM code to the upper RAM. The CoCo 3 does this automatically at boot time, but you have to do it yourself with a CoCo 2. The two sets of results were again exactly the same as before.

A Request for clarification

On 2021/08/15, I sent the following email to Walter Zydhek at the wzydhek@internetcds.com email address published in the Unravelled books:

> Dear Mr. Zydhek –
>
> I have a question about some particular terminology in the books; a question which might be possible to quickly answer -or- which might take some time to consider.
>
> Specifically, it concerns the wording at addresses $FFDE and $FFDF.
>
> Would you have time to tell me how you understand those two lines (except for the line numbers, they appear to be identical in all four books)?

In that email, I was purposely vague, so as not to influence Mr. Zydhek's potential response in any way.

On 2021/08/20, the email message bounced back as undeliverable.

Results

A POKE into $FFDE sets RAMROM mode with:

```
Addresses $0000 to $7FFF = RAM
Addresses $8000 to $FEFF = ROM
Addresses $FF00 to $FFFF = Hardware Registers and
                             Interrupt Vectors
```

A POKE into $FFDF sets ALLRAM mode with:

```
Addresses $0000 to $FEFF = RAM
Addresses $FF00 to $FFFF = Hardware Registers and
                             Interrupt Vectors
```

Conclusions and Future Work

Either I've misunderstood the author's intent, or there's an error in the Unravelled books at addresses $FFDE and $FFDF. In either case, the reality is that:

A POKE or other write into $FFDE sets RAMROM mode, and

A POKE or other write into $FFDF sets ALLRAM mode.

I see no reason for any additional work in this matter, except for others to perhaps repeat my investigation to verify and confirm (or perhaps refute) my results.

Appendix A: Book Entries

From *Color Basic Unravelled II*, page A9:

```
0661 FFDE ROMCLR     RMB 1     ROM DISABLED
0662 FFDF ROMSET     RMB 1     ROM ENABLED
```

From *Extended Basic Unravelled II*, page A15:

```
1055 FFDE ROMCLR     RMB 1     ROM DISABLED
1056 FFDF ROMSET     RMB 1     ROM ENABLED
```

From *Disk Basic Unravelled II*, page A15:

```
1055 FFDE ROMCLR     RMB 1     ROM DISABLED
1056 FFDF ROMSET     RMB 1     ROM ENABLED
```

From *Super Extended Basic Unravelled II*, page A15:

```
1055 FFDE ROMCLR     RMB 1     ROM DISABLED
1056 FFDF ROMSET     RMB 1     ROM ENABLED
```

To me, this seemed to be clearly saying that I should store a value into $FFDE to disable the ROM and go into ALLRAM mode; and, conversely, I should store a value into $FFDF to enable the ROM, i.e. to go into RAMROM mode:

where I'm defining ALLRAM mode to mean addresses $0000 through $FEFF are all RAM,

and, where I'm defining RAMROM mode to mean addresses $0000 through $7FFF are RAM while addresses $8000 through $FEFF are ROM,

and, where, in both the ALLRAM mode and in the RAMROM mode, addresses $FF00 through $FFFF are hardware registers and interrupt vectors.

But this seemed reversed from what I had learned from others, and from personal experience. To me, it seems that the Unravelled books should instead read:

```
FFDE ROMSET     RMB 1     ROM ENABLED
FFDF ROMCLR     RMB 1     ROM DISABLED
```

or, in the active, rather than passive, voice:

```
FFDE ROMSET     RMB 1     TO ENABLE ROM
```

```
FFDF ROMCLR     RMB 1     TO DISABLE ROM
```

——

Now, it may be that I'm simply misinterpreting what is written in the Unravelled books. But, if so, then new CoCo programmers might be just as confused by the wording as I am. In which case, perhaps this paper will help keep at least one perplexing bug from invading their code.

Appendix B: My 1990 CF83 POLCAT Code

When I wrote the CF83 operating system back in 1990, it ran primarily in ALLRAM mode. However, it jumped out of ALLRAM and into RAMROM mode in order to process input from the keyboard.

```
00001 *************************
00002 *
00003 * POLCAT.ASM
00004 * MDJ 09-25-90
00005 *
00006 * POLLS THE KEYBOARD FOR
00007 * A KEYSTROKE
00008 *
00009 *************************
00010 *
00011 * U AND S STACKS MUST
00012 * ALREADY BE INITIALIZED
00013 *
00014 * NO ENTRY CONDITIONS
00015 *
00016 * ON RETURN:
00017 *  IF NO KEY PRESSED -
00018 *   CC FLAG Z = 1
00019 *   REGISTER A = 0
00020 *  IF A KEY WAS PRESSED -
00021 *   CC FLAG Z = 0
00022 *   REGISTER A = KEY
00023 *
00024 * ANY ROUTINE CALLING
00025 * THIS MUST PSHS A,CC
00026 * BEFORE CALLING AND
00027 * PULS A,CC ON RETURN
00028 *
00029 *************************
00030 *
00031 RAMROM  EQU     $FFDE   RAM/ROM MODE
00032 ALLRAM  EQU     $FFDF   ALL RAM MODE
00033 XPOLCT  EQU     $A000   ROM POLCAT JUMP ADDRESS
00034           ORG     $3F9B
00035 POLCAT  PSHS    Y,U,DP
00036           STA     RAMROM  SET RAM/ROM MODE
00037           JSR     [XPOLCT] GO TO ROM POLCAT
00038           STA     ALLRAM  SET ALL RAM MODE
```

```
00039          PULS    Y,U,DP
00040 ZEND     RTS
00041          END
```

As can be seen from lines 31 and 32, this code is the exact opposite of what seems to be published in the Unravelled books. And, this code has worked correctly for 30+ years now.

Appendix C: Prompt Attention

In the July, 1987 issue of *The Rainbow*, in an article entitled "Prompt Attention" on page 97, Joseph Forgione wrote:

> Are you tired of the color-changing cursor and OK prompt on your CoCo? Conversion will allow you to change it
>
> .
>
> Upon running, the OK prompt will be replaced by "Ready. " However, for Conversion to work properly, you must be in the "all-RAM " mode. This means your CoCo must have all the ROM copied into RAM and be running entirely from RAM memory. Some DOSs have special commands allowing this, and the Coco 3 is always in the RAM mode. But if you are using a standard Radio Shack CoCo I or 2 system, you can enter the RAM mode by first running Listing 1, **DRIVER**. …
>
> **Listing 1 : DRIVER**

```
1 DATA 26,80,142,128,0,127,255,2
22,166,132,127,255,223,167,132,4
8,1,140,255,0,38,239,28,159,57
2 FORA=&HE00 TO &HE18:READX:POKE
A,X:NEXTA:EXEC3584:POKE65503,0:P
RINT"OS IS NOW IN RAM!" …
```

For our purposes, the most interesting part of this code appears in the right half of the next-to-last line where we find the statement "**POKE65503,0**".

Expressed in hexadecimal, this is "**POKE &HFFDF,&H00" –**

which Forgione thus holds will, in my terms, disable ROM and enter ALLRAM mode; also the exact opposite of what the Unravelled books seem to be saying.

Please note that, before trying to run this code in VCC, I "uncrunched" it to look like this (see Appendix I for why):

```
1 DATA 26,80,142,128,0,127,255
2 DATA 222,166,132,127,255,223
3 DATA 167,132,48,1,140,255,0
4 DATA 38,239,28,159,57
5 FOR A = &HE00 TO &HE18
6 READ X
7 POKE A,X
8 NEXT A
```

```
9 EXEC 3584
10 POKE 65503,0
11 PRINT "OS IS NOW IN RAM!"
32767 END
```

Also note that Forgione's code primarily uses decimal numbers. It could be rewritten in completely hexadecimal format as:

```
1 DATA 1A,50,8E,80,00,7F,FF,DE,
A6,84,7F,FF,DF,A7,84,30,
01,8C,FF,00,36,EF,1C,9F,39
2 FORA=&H0E00 TO &H0E18:READX$:POKE
A,VAL("&H"+X$):NEXTA:EXEC&H0E00:
POKE&HFFDF,&H00:PRINT"OS IS NOW IN RAM!"
```

Or, when uncrunched:

```
1 DATA 1A,50,8E,80,00,7F,FF,DE
2 DATA A6,84,7F,FF,DF,A7,84,30
3 DATA 01,8C,FF,00,26,EF,1C,AF
4 DATA 39
5 FOR A = &H0E00 TO &H0E18
6 READ X$
7 POKE A,VAL("&H"+X$)
8 NEXT A
9 EXEC &H0E00
10 POKE &HFFDF,&H00
11 PRINT "OS IS NOW IN RAM!"
32767 END
```

Poking machine language into memory like this is an established CoCo tradition, it saves column space when published in a magazine, and it avoids forcing the user to learn assembly language.

But, it also makes it nearly impossible to understand what the code is actually doing. Disassembling line 1 yields:

```
00100 *****
00110 *
00120 * FORGR2R.ASM
00130 * MDJ 2021/08/18
00140 *
00150 * COPIES ROM TO
00160 * UPPER RAM
00170 *
00180 * THIS IS A DISASSEMBLY
00190 * OF FORGIONE'S VERSION
00200 * OF A ROM TO RAM
```

```
                    00210 * PROGRAM
                    00220 *
                    00230 *****
                    00240
0E00                00250         ORG     $0E00
                    00260
                    00270 * SET THE IRQ AND FIRQ
                    00280 * INTERRUPT MASK BITS
0E00 1A   50        00290         ORCC    #$50
                    00300
                    00310 * POINT TO START OF ROM
0E02 8E   8000      00320         LDX     #$8000
                    00330
                    00340 * SET RAMROM MODE
0E05 7F   FFDE      00350 LBLA    CLR     $FFDE
                    00360
                    00370 * COPY ONE BYTE FROM ROM
0E08 A6   84        00380         LDA     ,X
                    00390
                    00400 * SET ALLRAM MODE
0E0A 7F   FFDF      00410         CLR     $FFDF
                    00420
                    00430 * PASTE ONE BYTE TO RAM
0E0D A7   84        00440         STA     ,X
                    00450
                    00460 * INCREMENT ADDRESS
                    00470 * POINTER
0E0F 30   01        00480         LEAX    1,X
                    00490
                    00500 * HAVE WE REACHED THE
                    00510 * I/O REGISTERS?
0E11 8C   FF00      00520         CMPX    #$FF00
                    00530
                    00540 * GO IF NO
0E14 26   EF        00550         BNE     LBLA
                    00560
                    00570 * CLEAR THE FIRQ
                    00580 * INTERRUPT MASK BIT
                    00590 * AND THE HALF-CARRY BIT
0E16 1C   9F        00600         ANDCC   #$9F
                    00610
                    00620 *****
                    00630 *
                    00640 * NOTE POSSIBLE ERROR
                    00650 * IN LINE 600
                    00660 *
                    00670 * I BELIEVE THIS SHOULD
```

```
                    00680 * BE:
                    00690 *
                    00700 * CLEAR IRQ AND FIRQ
                    00710 * INTERRUPT MASK BITS
                    00720 *        ANDCC   #$AF
                    00730 *
                    00740 *****
                    00750
                    00760 * EXIT
0E18 39             00770          RTS
          0000      00780          END

00000 TOTAL ERRORS
```

Compare this code with that of Huffman and Castro in Appendix H.

...Appendix D: Changing the Language

In the June, 1988 issue of *The Rainbow*, in an article entitled “Changing the Language” on page 168, Marc Campbell wrote:

> [Y]our CoCo must first be in "all-RAM " mode. This simply means that everything the computer stores in ROM (Read-Only Memory, the portion of a computer's memory that cannot be written over) must be copied into RAM (Random Access Memory, memory that both the user and the computer can read and/or revise). A short machine language routine by Joseph Forgione ("Prompt Attention, " THE RAINBOW, July '87, Page 97) does the trick quite nicely and is found in Listing 1 . On certain machines, you may experience a lock-up when you run Listing 1. If so, precede POKE65503,0 with EXEC &HE00 . . (If you happen to own a CoCo 3, then you've lucked out again because it is always in all-RAM mode.)

Again, **`65503`** is **`&HFFDF`** in hexadecimal, and thus Campbell is also, in my terms, indicating that a write into $FFDF will disable ROM and enter ALLRAM mode; the exact opposite of what the Unravelled books seem to be saying.

Appendix E: CoCo Hardware Programming

On page 72 of "Color Computer 1/2/3 Hardware Programming", Chris Lamont presents the following:

$FFDE/$FFDF (65502/65503) ROM/RAM map type - SAM_TYP

$FFDE Any write switches system ROMs into memory map (ROM mode)
$FFDF Any write selects all-RAM mode (RAM mode).

Here, Lamont directly contradicts what the Unravelled books seem to be saying.

Appendix F: SYSTST01 Code

To test the apparent tenets of the Unravelled books, I recently wrote **SYSTST01.BAS** to follow the directives as I understood them to be written therein:

```
1000 '*****
1010 '*
1020 '* SYSTST01.BAS
1030 '* MDJ 2021/08/15
1040 '*
1050 '*
2000 '
2010 PRINT "SYSTEM TEST 01"
2030 '
2100 '*****
2110 '*
2120 '* INITIAL RAMROM TEST
2130 '*
2140 '* CF. DISK BASIC
2150 '*     UNRAVELLED
2160 '*       PAGE A15
2170 '*
2180 '*****
2190 '
2200 'SET RAMROM MODE
2210 PRINT "SETTING RAMROM: ";
2220 POKE &HFFDF, 0
2230 GOSUB 3500
2240 PRINT
2250 '
2300 '*****
2310 '*
2320 '* SECOND RAMROM TEST
2330 '*
2380 '*****
2390 '
2400 'SET ALLRAM MODE
2410 PRINT "SETTING ALLRAM: ";
2420 POKE &HFFDE, 0
2430 GOSUB 3500
2440 PRINT
2450 '
2500 '*****
```

```
2510 '*
2520 '* THIRD RAMROM TEST
2530 '*
2580 '*****
2590 '
2600 'SET RAMROM MODE
2610 PRINT "SETTING RAMROM: ";
2620 POKE &HFFDF, 0
2630 GOSUB 3500
2640 PRINT
2650 '
2660 'END OF RUN
2670 GOTO 32767
2680 '
3000 '*****
3010 '*
3020 '* CHECK ALLRAM/RAMROM
3030 '* STATUS
3040 '*
3050 '* &HBFE6 THROUGH &HBFEF
3060 '* ARE TEN UNUSED GARBAGE
3070 '* BYTES AT THE END OF
3080 '* COLOR BASIC ROM, JUST
3090 '* BEFORE THE INTERRUPT
3100 '* VECTORS. THEIR HEX
3110 '* VALUES ARE:
3120 '*
3130 '*   A1 54 46 8F 13
3140 '*   8F 52 43 89 CD
3150 '*
3160 '* CF. COLOR BASIC
3170 '*     UNRAVELLED
3180 '*       PAGE B53
3190 '*
3200 '*****
3210 '
3220 'GET ROM JUNK VALUE
3500 V1 = PEEK(&HBFE6)
3510 '
3520 'INCREMENT IT
3530 V2 = V1 + 1
3540 '
3550 'PUT NEW VALUE
3560 POKE &HBFE6, V2
3570 '
3580 'GET CURRENT VALUE
3590 V3 = PEEK(&HBFE6)
```

```
3600 '
3630 PRINT "NOW IN ";
3640 '
3650 'GO IF CURRENT = NEW
3660 IF V3 = V2 GOTO 3750
3670 '
3680 'CURRENT <> NEW
3690 'IMPLIES RAMROM
3700 PRINT "RAMROM."
3710 GOTO 3770
3720 '
3730 'CURRENT = NEW
3740 'IMPLIES ALLRAM
3750 PRINT "ALLRAM."
3760 '
3770 'RESTORE ROM JUNK VALUE
3780 POKE &HBFE6, V1
3790 '
3800 PRINT "V1 = "; HEX$(V1);
3810 PRINT " V2 = "; HEX$(V2);
3820 PRINT " V3 = "; HEX$(V3);
3830 RETURN
3840 '
32767 END
```

Results:

```
SYSTEM TEST 01
SETTING RAMROM: NOW IN ALLRAM.
V1 = A1 V2 = A2 V3 = A2
SETTING ALLRAM: NOW IN RAMROM.
V1 = A1 V2 = A2 V3 = A1
SETTING RAMROM: NOW IN ALLRAM.
V1 = A1 V2 = A2 V3 = A2
```

As can be seen from the results above, if my understanding of what is written in the Unravelled books correctly reflects what the wording seems to intend, the actual results are indeed the exact opposite of that intention.

These results were obtained by running this program in the VCC emulator immediately after startup.

These exact same results were also obtained by running this program in the VCC emulator after setting it to CoCo 2 mode, i.e. by using the command sequence

```
X=PEEK(&HFF90):X=X OR &H80:POKE&HFF90,X:RUN.
```

These exact same results were also obtained by running this program in my physical 64K CoCo 2 Setup (See Appendix J). However, it was necessary to run the **ROM2RAM.BAS** program described in Appendix H before running this **SYSTST01.BAS** program in that physical setup.

In the CoCo 3, during the boot process, the ROM is copied into upper RAM and then the CoCo 3 is put into, and continues to run from, RAM. This doesn't happen during the CoCo 2 boot process: You thus have to copy the ROM into the upper RAM yourself.

Appendix G: SYSTST02 Code

I recently wrote **SYSTST02.BAS** to test my understanding of what is actually required to properly switch between the ALLRAM and RAMROM modes:

```
1000 '*****
1010 '*
1020 '* SYSTST02.BAS
1030 '* MDJ 2021/08/15
1040 '*
1050 '*
2000 '
2010 PRINT "SYSTEM TEST 02"
2030 '
2100 '*****
2110 '*
2120 '* INITIAL RAMROM TEST
2130 '*
2135 `* DOES THE OPPOSITE OF
2140 '*     DISK BASIC
2150 '*     UNRAVELLED
2160 '*       PAGE A15
2170 '*
2180 '*****
2190 '
2200 'SET RAMROM MODE
2210 PRINT "SETTING RAMROM: ";
2220 POKE &HFFDE, 0
2230 GOSUB 3500
2240 PRINT
2250 '
2300 '*****
2310 '*
2320 '* SECOND RAMROM TEST
2330 '*
2380 '*****
2390 '
2400 'SET ALLRAM MODE
2410 PRINT "SETTING ALLRAM: ";
2420 POKE &HFFDF, 0
2430 GOSUB 3500
2440 PRINT
2450 '
```

```
2500 '*****
2510 '*
2520 '* THIRD RAMROM TEST
2530 '*
2580 '*****
2590 '
2600 'SET RAMROM MODE
2610 PRINT "SETTING RAMROM: ";
2620 POKE &HFFDE, 0
2630 GOSUB 3500
2640 PRINT
2650 '
2660 'END OF RUN
2670 GOTO 32767
2680 '
3000 '*****
3010 '*
3020 '* CHECK ALLRAM/RAMROM
3030 '* STATUS
3040 '*
3050 '* &HBFE6 THROUGH &HBFEF
3060 '* ARE TEN UNUSED GARBAGE
3070 '* BYTES AT THE END OF
3080 '* COLOR BASIC ROM, JUST
3090 '* BEFORE THE INTERRUPT
3100 '* VECTORS. THEIR HEX
3110 '* VALUES ARE:
3120 '*
3130 '*   A1 54 46 8F 13
3140 '*   8F 52 43 89 CD
3150 '*
3160 '* CF. COLOR BASIC
3170 '*     UNRAVELLED
3180 '*       PAGE B53
3190 '*
3200 '*****
3210 '
3220 'GET ROM JUNK VALUE
3500 V1 = PEEK(&HBFE6)
3510 '
3520 'INCREMENT IT
3530 V2 = V1 + 1
3540 '
3550 'PUT NEW VALUE
3560 POKE &HBFE6, V2
3570 '
3580 'GET CURRENT VALUE
```

```
3590 V3 = PEEK(&HBFE6)
3600 '
3630 PRINT "NOW IN ";
3640 '
3650 'GO IF CURRENT = NEW
3660 IF V3 = V2 GOTO 3750
3670 '
3680 'CURRENT <> NEW
3690 'IMPLIES RAMROM
3700 PRINT "RAMROM."
3710 GOTO 3770
3720 '
3730 'CURRENT = NEW
3740 'IMPLIES ALLRAM
3750 PRINT "ALLRAM."
3760 '
3770 'RESTORE ROM JUNK VALUE
3780 POKE &HBFE6, V1
3790 '
3800 PRINT "V1 = "; HEX$(V1);
3810 PRINT " V2 = "; HEX$(V2);
3820 PRINT " V3 = "; HEX$(V3);
3830 RETURN
3840 '
32767 END
```

Results:

```
SYSTEM TEST 02
SETTING RAMROM: NOW IN RAMROM.
V1 = A1 V2 = A2 V3 = A1
SETTING ALLRAM: NOW IN ALLRAM.
V1 = A1 V2 = A2 V3 = A2
SETTING RAMROM: NOW IN RAMROM.
V1 = A1 V2 = A2 V3 = A1
```

As can be seen from the results above, this code correctly performs the intended switches between the ALLRAM and RAMROM modes.

These results were obtained by running this program in the VCC emulator immediately after startup.

These exact same results were also obtained by running this program in the VCC emulator after setting it to CoCo 2 mode, i.e. by using the command sequence

X=PEEK(&HFF90):X=X OR &H80:POKE&HFF90,X:RUN.

These exact same results were also obtained by running this program in my physical 64K CoCo 2 Setup (See Appendix J). However, it was necessary to run the **ROM2RAM.BAS** program described in Appendix H before running this **SYSTST02.BAS** program in that physical setup.

In the CoCo 3, during the boot process, the ROM is copied into upper RAM and then the CoCo 3 is put into, and continues to run from, RAM. This doesn't happen during the CoCo 2 boot process: You thus have to copy the ROM into the upper RAM yourself.

Appendix H: Huffman & Castro Code

As I mentioned in Appendices F and G, the simple code presented there works in the VCC emulator (and presumably also in a real, physical CoCo 3) because the upper RAM has already been populated with the ROM code at boot time.

The situation is different for a real, physical 64K CoCo 2. There, the ROM is not copied to RAM during boot, so you have to do the copy yourself.

In 2016, Allen Huffman, with an assist from Juan Castro, published a version of the classic **ROM2RAM** program which does exactly that; and also changes the CoCo prompt from **OK** to **OY** just to confirm that the program has worked. Their code is:

```
10 READ A$:IF A$="X" THEN 35
20 POKE 20000+N,VAL("&H"+A$)
30 N=N+1:GOTO 10
35 EXEC 20000:POKE &HABEF,89:END
40 DATA 34,01,1A,50,10,8E,80,00
50 DATA B7,FF,DE,EC,A4,AE,22,EE
60 DATA 24,B7,FF,DF,ED,A1,AF,A1
70 DATA EF,A1,10,8C,FE,FC,25,E8
80 DATA 10,8C,FF,00,24,0C,B7,FF
90 DATA DE,EC,A4,B7,FF,DF,ED,A1
100 DATA 20,EE,35,01,39
110 DATA X
```

Poking machine language into memory like this is an established CoCo tradition, it saves column space when published in a magazine, and it avoids forcing the user to learn assembly language.

But, it also makes it nearly impossible to understand what the code is actually doing. Disassembling lines 40 through 100, and noting that decimal 20000 = hexadecimal $4E20, yields:

```
00100 *****
00110 *
00120 * ROM2RAM.ASM
00130 * MDJ 2021/08/18
00140 *
00150 * COPIES ROM TO
00160 * UPPER RAM
00170 *
00180 * THIS IS A DISASSEMBLY
00190 * FROM ALLEN HUFFMAN AND
00200 * JUAN CASTRO'S 2016
```

```
                    00210 * VERSION OF ROM2RAM.BAS
                    00220 *
                    00230 *****
                    00240
4E20                00250         ORG     $4E20
4E20 34   01        00260         PSHS    CC
                    00270
                    00280 * SET THE IRQ AND FIRQ
                    00290 * INTERRUPT MASK BITS
4E22 1A   50        00300         ORCC    #$50
                    00310
                    00320 * POINT TO START OF ROM
4E24 108E 8000      00330         LDY     #$8000
                    00340
                    00350 * SET RAMROM MODE
4E28 B7   FFDE      00360 LBLA    STA     $FFDE
                    00370
                    00380 * COPY SIX BYTES
                    00390 * AT A TIME FROM ROM
4E2B EC   A4        00400         LDD     ,Y
4E2D AE   22        00410         LDX     2,Y
4E2F EE   24        00420         LDU     4,Y
                    00430
                    00440 * SET ALLRAM MODE
4E31 B7   FFDF      00450         STA     $FFDF
                    00460
                    00470 * PASTE SIX BYTES
                    00480 * AT A TIME TO RAM
4E34 ED   A1        00490         STD     ,Y++
4E36 AF   A1        00500         STX     ,Y++
4E38 EF   A1        00510         STU     ,Y++
                    00520
                    00530 * CLOSER THAN SIX BYTES
                    00540 * TO THE START OF
                    00550 * I/O REGISTERS?
4E3A 108C FEFC      00560         CMPY    #$FEFC
                    00570
                    00580 * GO IF NO
4E3E 25   E8        00590         BLO     LBLA
                    00600
                    00610 * LESS THAN START OF
                    00620 * I/O REGISTERS?
4E40 108C FF00      00630 LBLB    CMPY    #$FF00
                    00640
                    00650 * GO IF NO
4E44 24   0C        00660         BHS     LBLC
                    00670
```

```
                        00680 * SET RAMROM MODE
4E46 B7   FFDE          00690         STA     $FFDE
                        00700
                        00710 * COPY NMI JUMP VECTOR
                        00720 * FROM ROM
4E49 EC   A4            00730         LDD     ,Y
                        00740
                        00750 * SET ALLRAM MODE
4E4B B7   FFDF          00760         STA     $FFDF
                        00770
                        00780 * PASTE NMI JUMP VECTOR
                        00790 * TO RAM
4E4E ED   A1            00800         STD     ,Y++
                        00810
                        00820 * GO CHECK NEXT ADDRESS
4E50 20   EE            00830         BRA     LBLB
                        00840
                        00850 * EXIT
4E52 35   01            00860 LBLC    PULS    CC
4E54 39                 00870         RTS
          0000          00880         END

00000 TOTAL ERRORS
```

You can see that the bytes generated by assembling this disassembly do indeed match the bytes being **POKE**d into memory by the BASIC program.

As an aside, note that this code also confirms my contention that:

A POKE or other write into $FFDE sets RAMROM mode, and

A POKE or other write into $FFDF sets ALLRAM mode.

As another aside, I found lines 530 through 660 particularly interesting. The code here is just repeatedly doing a copy and paste operation from ROM to upper RAM, six bytes at a time. If we start at $8000 and count upwards by sixes, we eventually land on $FEFC exactly. If we then add another six, we get:

```
$FEFC + $0006 = $FF02
```

which is an over-run. We would wind up copying the first two bytes of the I/O registers in addition to the ROM code. We don't want to do that. So, we stop counting by sixes, and count by twos instead.

```
$FEFC + $0002 = $FEFE
$FEFE + $0002 = $FF00
```

and we stop right on target exactly. Neat!

When I ran this code on my physical 64K CoCo 2 setup, and then ran **SYSTST01.BAS** and SYSTST02.BAS, I got the exact same results as I did when I ran those two tests in VCC.

Compare this code with that of Forgione in Appendix C.

Appendix I: Clarity vs. Memory Space

Let's take a look at Joseph Forgione's code with respect to clarity and memory space:

```
1 DATA 26,80,142,128,0,127,255,2
22,166,132,127,255,223,167,132,4
8,1,140,255,0,38,239,28,159,57
2 FORA=&HE00 TO &HE18:READX:POKE
A,X:NEXTA:EXEC3584:POKE65503,0:P
RINT"OS IS NOW IN RAM!"
```

To be polite, let's just say that clarity is not its strongest point. But, let's not dismiss its appearance out-of-hand without first thinking about why it is what it is.

First of all, we might be misled into thinking there are six lines in the above code. Actually, there are only two. When typing into a CoCo's 32-character-wide screen, we can enter long lines by just continuing to type. There appears to be a second line on the screen, but it's really just a continuation of the first line. And that can continue to "third" lines and beyond.

This may be hard to conceive in Forgione's code because we can't easily examine the full-length line for comparison. The first line is 94 characters long, and the second line is 87 characters long. If expanded, neither line would fit on this page.

But, suppose the screen was ten characters wide instead of 32. And, suppose we typed-in a long line of code like this:

```
2 FORA=&HE
00 TO &HE1
8:READX:PO
KEA,X:NEXT
A
```

When not limited by the screen's ten character width, the expanded line would look like this:

```
2 FORA=&HE00 TO &HE18:READX:POKEA,X:NEXTA
```

Comparing the "crunched" code with the "uncrunched" gives us a picture of what's actually going on.

Although clarity is not one of them, there **WERE** valid and important reasons for using crunched code instead of uncrunched. Among them were:

1. They fit magazine column width well.
2. They minimized the expenditure of magazine column length.
3. They made it easier to catch typing errors when you were entering the code, because the width in the magazine exactly matched the width of the screen, and many kinds of typos would stand out rather glaringly.
4. They minimized memory space usage.

Reason 4 was quite critical, especially on the early 4K CoCo 1 (my first CoCo).

For example, Forgione's code, as presented above, occupies 94 + 87 + 2 = 183 bytes (The "2" is for the invisible carriage return bytes at the end of each line). Now consider a just slightly "clarified" version:

```
1 DATA 26,80,142,128,0,127,255
2 DATA 222,166,132,127,255,223
3 DATA 167,132,48,1,140,255,0
4 DATA 38,239,28,159,57
5 FOR A = &HE00 TO &HE18
6 READ X
7 POKE A,X
8 NEXT A
9 EXEC 3584
10 POKE 65503,0
11 PRINT "OS IS NOW IN RAM!"
32767 END
```

(I always make sure every BASIC program has an END statement). This code, including carriage returns, occupies 237 bytes, a 30% increase.

But, except in extra-long, extra-critical "Stuff it til it explodes!" code, this byte-saving is not as crucial on a 64K CoCo 2; certainly not with short little programs like this.

So, as I do in this paper, I tend to expand such published code into more clarified presentations like the above.

Also, there's one other little GOTCHA waiting for you if you try to use crunched code in VCC. VCC 2.1.0d includes a very nice "Edit" menu which allows you to copy and paste text and code between VCC and other documents from other programs running in Windows.

For example, if you're reading this paper in Adobe Acrobat Reader (or reading some other document in Word or Notepad), you can copy the above code into the Windows System Clipboard from within Reader, and then just paste it into VCC using VCC's Edit. REALLY saves on typing time!

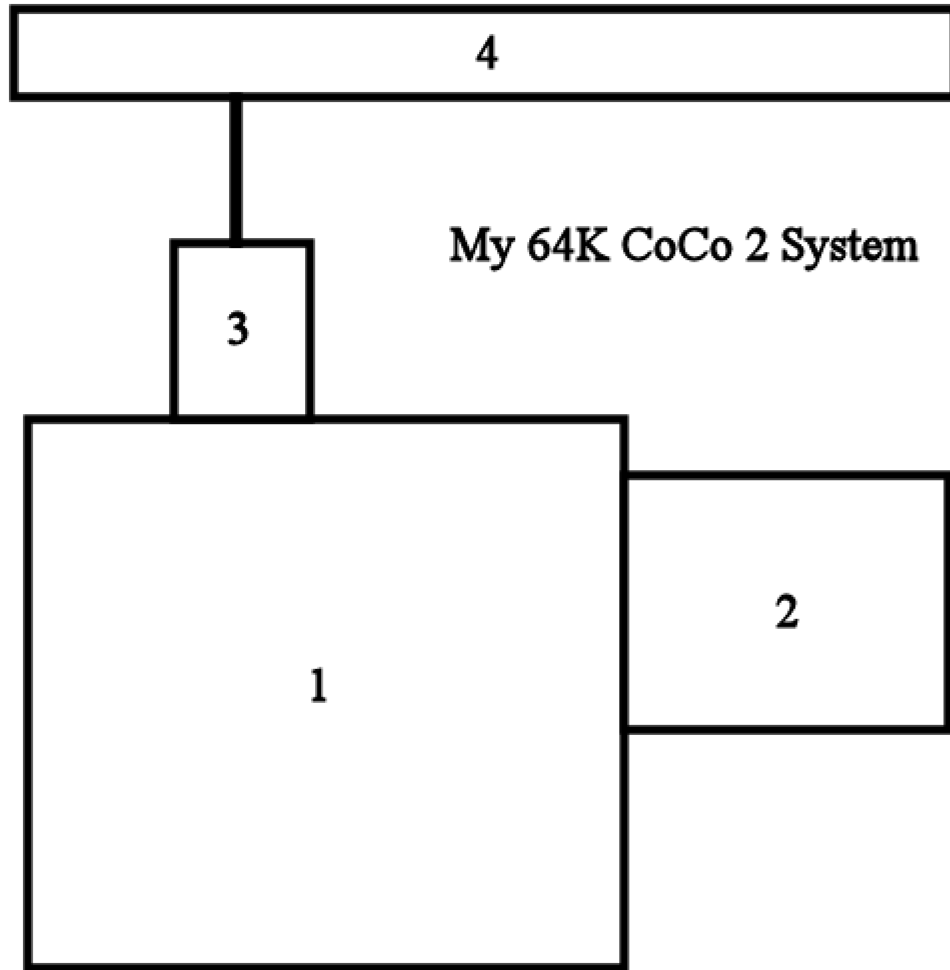
However, just as Forgione's crunched code initially seemed to be six lines long to us, VCC tends to see it as six lines too, because, in the document you're copying from, it actually is six lines. In Adobe Reader, and in Word, and in Notepad, there's a hidden carriage return at the end of each of those six pseudo-lines. And when you try to paste them into VCC, VCC has a fit and throws "SN ERROR" messages all over the place.

Which is another good reason for expanding crunched code the way I do here.

One other little gotcha: Be mindful of where you're copying from when you're pasting into VCC. I had no problem pasting from Adobe, Word, or Notepad, but when I tried to paste from Scrivener (manuscript preparation software) the quotation marks got lost and didn't show up in VCC. I don't know what Scrivener was doing to them, but pasting the code from Scrivener into Notepad, and then from Notepad into VCC solved the problem.

Appendix J: My 64K CoCo 2 System

My physical 64K CoCo 2 System is as depicted in the block diagram below.



1. 64K CoCo 2, Model Number 26-3127, Serial Number 007601.
2. CoCo SDC, running SDC-DOS 1.6 CC2.
3. CoCo VGA.
4. RCA 19" VGA/HDMI Monitor, Model RT1970.

Appendix K: My CoCo Philosophy

The CoCo community enjoys a great diversity of interests.

Some choose to concentrate on hardware innovations and modifications such as interfacing with VGA and HDMI monitors, SD Card data storage, and 104-key keyboards. This interest is at least partly born of necessity, since composite monitors, floppy diskettes, and CoCo spare parts are no longer manufactured and are in increasingly short supply.

Others concentrate on expanding the software horizons of the CoCo 3, using NitrOS-9 and other operating systems to make the multitasking CoCo behave ever closer to modern Windows, Mac, and Linux machines.

Still others are devoted to emulating the CoCo on other platforms by developing emulators such as VCC, OVCC, MAME, and XRoar.

And some just love retro gaming.

My personal interest is twofold:

1. To see VCC increasingly used as a learning tool for budding software developers.

2. To see just how much I can cram into a 64K CoCo 2.

First, VCC: Today's Grade School, Junior High, and High School students have a wealth of available learning tools. Micro-bits, Arduinos, and Raspberry Pi supermicro devices provide highly affordable entry-level introductions to computer programming and interfacing. Maker-Spaces and Innovation Centers in our schools and libraries help foster growth and experience.

But these devices do have limitations. Even these simple(?) computers can have rather steep learning curves, and their low initial cost can quickly expand as new peripherals and experimental equipment and supplies are added.

VCC is free, and can be used on any Windows computer: just download it, install it, and it runs. If you don't own a Windows computer, your school, library, or a friend probably does. The included BASIC language is easy to learn and can readily serve as a stepping-stone towards more complex programming languages. (And, no, learning structured programming does not require a language that enforces structure. In fact, I think learning to structure your programs is actually more effective when you do so on your own.)

I prefer VCC to the other emulators for these purposes because its setup is trivial: Again, just download it, install it, and it runs. OVCC, MAME, and XRoar have their advantages, but ease of setup is

not one of them. Even with their available Windows binary packages, they require pre-installation of other bits and pieces of software before they can be downloaded, installed, and run. This may not be a major problem for a reasonably adept aficionado, but it forms a significant barrier for the newbie. And, it's the newbie whom we're trying to reach, interest, and encourage here; the newbie who may not yet recognize even the tiniest awakening of interest in things computational.

But, for these purposes, VCC has one glaring weakness: its instruction manual is woefully terse. I would like to see VCC bundled with a selection of tutorials, manuals, and examples suited to guiding even the most newbie of newbies into the wonders of computing.

Second, The Stuffed CoCo: I'm simply fascinated by the challenge of seeing how much functional capability I can sandwich into the nooks and crannies of the 64K space. Whether it's working in the available RAM left by the 32K ROM and the dedicated RAM that supports that ROM, or whether it's jumping right into ALLRAM mode and just filling the entire 64K to near-overflowing; it's an investigative gauntlet which goes right to the heart of my enchantment with puzzles in general.

It's great fun!

M.D.J. 2021/08/29

Appendix L: New BDS Software License

This New Software License applies to all software found on the BDS Software site, and supersedes all previous copyright notices and licensing provisions which may appear in the software itself or in any documentation therefor.

All software which has previously been placed in the public domain remains in the public domain.

All other software, programs, experiments and reports, documentation, and any other material on this site (other than that attributed to outside sources) is hereby copyright © 2018 (or later if so marked) by M. David Johnson.

All software, documentation, and other information on the BDS Software site is available for you to freely download without cost.

Whether you downloaded such items directly from this site, or you obtained them by any other means, you are hereby licensed to copy them, to sell or give away such copies, to use them, and to excerpt from them, in any way whatsoever, so long as nothing you do with them would denigrate the name of our Lord and Savior, Jesus Christ.

I make absolutely no warranty whatsoever for any of these items. You use them entirely at your own risk.

If they don't work for you, I commiserate.

If they crash your system, I sympathize.

But I accept no responsibility whatsoever for any such consequences. Under no circumstances will BDS Software or M. David Johnson be liable for any negative results of any kind which you may experience from downloading or using these items.

BDS Software's former mail address at P.O. Box 485 in Glenview, IL is no longer valid. Any mail sent to that address will be rejected by the U.S. Postal Service. See my Contact page.

M.D.J. 2018/06/08

Works Cited

Campbell, Marc. “Changing the Language.” *The Rainbow*. 168. Prospect, KY: Falsoft, 1988/06. Print.

Forgione, Joseph. “Prompt Attention.” *The Rainbow*. 97. Prospect, KY: Falsoft, 1987/07. Print.

Huffman, Allen and Castro, Juan. “64K TRS-80 CoCo memory test.” *Sub-Etha Software*, https://subethasoftware.com/2016/01/19/64k-trs-80-coco-memory-test/ . Web. Last Accesses 2021/08/18.

Johnson, M. David. “The CF83 System.” *BDS Software*. http://www.bds-soft.com/php/coco/CF83System.php . Web. Last Accessed 2021/08/18.

Lomont, Chris. “Color Computer 1/2/3 Hardware Programming, version 0.82.” *Lamont.org* . http://www.lomont.org/software/misc/coco/Lomont_CoCoHardware.pdf . Web. Last Accessed: 2021/08/15.

Motorola. *“SN74LS783 MC6883 Synchronous Address Multiplexer* (SAM) Datasheet.” *Color Computer Archive*. https://colorcomputerarchive.com/repo/Documents/Datasheets/MC6883%20Synchronous%20Address%20Multiplexer%20Advance%20Information%20(Motorola).pdf . Last Accessed 2021/08/15.

Zydhek, Walter. *Color Basic Unravelled II*. Internet Archive. https://archive.org/details/Color_Basic_Unravelled_II_1999_Spectral_Associates .Last Edited 2013/05/20. Web. Last Accessed 2021/08/15.

Zydhek, Walter. *Disk Basic Unravelled II*. Internet Archive. https://archive.org/details/Disk_Basic_Unravelled_II_1999_Spectral_Associates .Last Edited 2013/05/20. Web. Last Accessed 2021/08/15.

Zydhek, Walter. *Extended Color Basic Unravelled II*. Internet Archive. https://archive.org/details/Extended_Basic_Unravelled_II_1999_Spectral_Associates .Last Edited 2013/05/20. Web. Last Accessed 2021/08/15.

Zydhek, Walter. *Super Extended Color Basic Unravelled II*. Internet Archive. https://archive.org/details/Super_Extended_Basic_Unravelled_II_1999_Spectral_Associates .Last Edited 2013/05/20. Web. Last Accessed 2021/08/15.

Other References

Barden Jr., William. *TRS-80 Color Computer Assembly Language Programming*. Fort Worth, TX: Radio Shack, 1983. Print.

Pierce, Bill et al. “Welcome to VCC (Virtual Color Computer, Version 2.1.0d.” *GitHub*. https://github.com/VCCE/VCC/releases Last Edited 2021/07/03. Web. Last Accessed 2021/08/20.

Leventhal, *Lance. 6809 Assembly Language Programming*. Berkeley, CA: Osborne/McGraw Hill. 1981. Print.

Microsoft. *Disk EDTASM Color Computer Disk Editor Assembler with ZBUG*. Fort Worth, TX: Radio Shack. 1983. Print.

Warren, Carl. *MC6809 Cookbook, The*. Blue Ridge Summit, PA: TAB Books, 1980. Print.