The Curious Case of Coleco Copy Control

I was talking to a friend about the game Risky Rick- a new homebrew for the Colecovision produced by Arcadevision. Specifically, he dumped the game so he could play it on his Atarimax flash cart without having to put wear on the original cartridge. After dumping, the game worked but would mysteriously prevent you from completing level 1. You could play through level 1, and at the end where you crawl through a small gap to the next level, the game will not let you progress at the edge of the screen.

Checking the Atariage forum thread about this game revealed that several people have had this same mysterious bug with the game... even playing the legit cartridge. Arcadevision's reply was simply the game is not designed to work on "modified" Coleco systems, and those people were basically on their own.

After many pages of back and forth between people who couldn't get the game they purchased to work properly on their Colecovisions, someone discovered that if pin 13 on the cartridge was grounded, the game would work!

Reading the thread some more , Arcadevision declared that the game had no DRM, and it was just the "modified" systems' fault. Not to pass up a detective story this good, I obtained the current ROM dump, I promptly disassembled it and went through the code.

On first inspection, the game appeared to have no copy protection, detection routines, mappers, or anything like that. It was just a flat 32K ROM image. Sure enough, you get stuck at the end of level 1. Interestingly, there's quite a bit of what looks like wasted space near the end of the ROM with a repeated data structure.

At this point, I smelled a rat and figured everything was wrapped up in pin 13 of the cartridge. Checking a pinout here: http://mheironimus.blogspot.com/2014/12/arduino-colecovision-cartridge-reader.html reveals that pin 13 is a ground. The schematic for the Colecovision shows that this isn't really a true ground- it's a "shield ground". It is only connected to ground through the metal shielding that covers the PCB.

The commercial Coleco cartridges do not connect pin 13 on the cartridge at all- pin 29 is the actual circuit ground which is connected to the chips on the cartridge. This means, the game could be using pin 13 to determine if you had an "modified" system or not!

Pin 13 was measured with the power on, and showed 5V. This means the pin is connected! A normal cartridge would show nothing here. After grounding pin 13 and redumping, the data was totally different from the first original dump, confirming that there are two ROMs in the one cartridge- the "demo" version of the game, and the "full" version of it.

This explains why people who did the "5V RAM" modification to their Coleco to fix the VRAM had issues with the game. The mod itself was a red herring, and the real reason was they replaced the RAMs and didn't bother to put the shielding back on. I don't blame them, either. It's a big pain to remove and install it. At least one person had a stock Coleco that was untouched, and the ground tab from the shield was bent so it didn't make contact, and only being able to play the demo is the result.

Apparently, the common Coleco cartridge dumpers do not connect pin 13, which means when dumped this way, you will get the demo ROM and not the full game. Grounding pin 13 on the dumper fixes that problem.

Once the “real” game’s ROM image was obtained, it did not work on the Atarimax cartridge, or emulators. A disassembly of the new ROM showed that there was now some extra code added to the front before the game code itself! At this point, I knew Arcadevision was lying about not including DRM with the game.

The TL/DR version is that there’s a routine that first detects emulators by testing RAM. It does this by checking adjacent bytes of RAM. If there are 256 or more adjacent bytes, the game will crash/lock up. Interestingly, hitting reset will bypass this test- it only does this check once on reset.

The second check is a bit more insidious, and I know at least one person where this test falsely flags their Colecovision as unclean, and the game won’t run at all. The check works by testing the Coleco RAM at boot for 8 consecutive ASCII characters. If they are found, it will crash/lock up. This works by checking each byte of RAM for the value 0x20-0x7b or so. If 8 consecutive bytes are in this range, it’s game over.

This test is pretty poor because there can be a high chance of 8 bytes being in this range in order somewhere in the 1024 bytes of RAM. I am fairly convinced this test is specifically for the Atarimax flash cartridge. The game refuses to run on it no matter what.

If you wish to bypass all the copy protection, change byte 0x000a in the ROM from 0x25 to 0x74.

Conclusion: Arcadevision released a homebrew that had three forms of DRM in it, while denying it had any. The pin 13 trick is absolutely brilliant and I have to give them credit for figuring that out. The unfortunate thing is now that the cat’s out of the bag, it was a one shot deal and that DRM will never work again. Two of the software checks were kind of a dick move and can get triggered even on legit systems.

If you’re interested in the software protections, here’s an in-depth.

On the demo version of the game, the game code starts at address 8025:

```
8025: CD 83 83 CALL 8383
8028: 21 00 00 LD HL,0000
802B: 11 00 40 LD DE,4000
802E: 3E 00 LD A,00
8030: CD 82 1F CALL 1F82
8033: 21 00 20 LD HL,2000
8036: 11 20 00 LD DE,0020
8039: 3E F0 LD A,F0
803B: CD 82 1F CALL 1F82
803E: CD 85 1F CALL 1F85
8041: CD 7F 1F CALL 1F7F
```

This has a pretty well defined structure, calling some BIOS routines to clear memory and such. We can see this same code on the full version of the game, but it's shifted down in memory.

```
8074: CD D2 83 CALL 83D2
8077: 21 00 00 LD HL,0000
807A: 11 00 40 LD DE,4000
807D: 3E 00 LD A,00
807F: CD 82 1F CALL 1F82
8082: 21 00 20 LD HL,2000
8085: 11 20 00 LD DE,0020
8088: 3E F0 LD A,F0
808A: CD 82 1F CALL 1F82
808D: CD 85 1F CALL 1F85
8090: CD 7F 1F CALL 1F7F
```

As you can see, the code has been shifted down 0x4f bytes vs. the demo.

The protection code is in two parts. There's some fancy footwork to calculate an address and push it on the stack for use later. This is just pure obfuscation, and pretty poor obfuscation at that.

The BIOS code looks like this starting at the reset vector:

```
 A0000 LD SP,Stack ; Initialize stack pointer
x JR L006E ; Go to rest of cold-start code
L006E LD HL,(0x8000) ; Check first word of cart for 55AAH
x LD A,L ; 8000=55H and 8001=AAH
x CP 55H
x JR NZ,L0081
x LD A,H
x CP 0AAH
x JR NZ,L0081
x LD HL,(0x800A) ; If 55H/AAH, jump into cartridge
x JP (HL)
```

Then the protection code start:

```
 8024: 11
8025: 11 2A 00 LD DE,002A
8027: 00 NOP
8028: 19 ADD HL,DE
8029: E5 PUSH HL ;add 002a to 8025, store onto stack (804f)
802A: D9 EXX
802B: 3A 24 80 LD A,(8024)
802E: 21 FF 73 LD HL,73FF ;check 11 from 8024 against a RAM byte at 73FF
8031: BE CP A,(HL)
8032: C8 RET Z ;if they are the same, jump to 804f
```

Address 0x800A holds 8025, which is where execution will start. The first thing the code does is adds 0x2A to the HL register, which is set to 0x8025 in the BIOS routine that points HL to the code start. This means HL will hold 0x804F. This is pushed onto the stack.

The EXX is just misdirection, and not used by anything. Next, HL is loaded with the last byte of RAM, which is used to determine if this is a warm or cold boot of the

system. The byte is read from ROM (which is 0x11, from address 8024) and then compared against the last RAM byte.

If the value matches, the RET Z is run, which jumps to 0x804F and skips the first protection check. Otherwise, execution continues at 0x8033 for the first check:

```
 8033: 77 LD (HL),A ;not the same, save it to RAM
8034: 01 00 04 LD BC,0400 ;bytes to test
8037: 21 00 70 LD HL,7000 ;RAM pointer
803A: 11 00 00 LD DE,0000 ;detection count
803D: 7E LD A,(HL) ;get a byte of RAM
803E: 23 INC HL ;increment RAM pointer
803F: BE CP A,(HL) ;compare to the next byte of RAM
8040: 20 01 JR NZ,8043 ;if the bytes do not match, skip the INC DE
8042: 13 INC DE ;increase detection count
8043: 0B DEC BC
8044: 78 LD A,B ;OR B and C registers together, this sets us up to check BC for 0
8045: B1 OR A,C
8046: 7A LD A,D ;put D into A
8047: 20 F4 JR NZ,803D ;we test all 1024 bytes.
8049: B2 OR A,D ;A holds the upper byte of the detection count. OR with itself will
check it for 0
804A: C8 RET Z ;if 255 or fewer matches, jump to 0x804F by using RET
804B: 10 E7 DJNZ 8033 ;fall through otherwise.. this will wedge/crash
804D: C5 PUSH BC
804E: C9 RET
```

The first thing that happens is 0x11 is written to the last byte of RAM, so next time the system is run, this first check will be skipped. This means there's a 1 in 256 chance that a typical Coleco will never run this check, or only run it some times.

Next, BC is loaded with the byte count (1024 bytes), HL is loaded with the start of RAM, and DE is cleared. DE holds the "match count".

Next, the first two bytes of RAM are compared against each other. If they match, DE is incremented. The next two bytes are tested, and so on. All 1024 bytes are tested, then the upper byte of the detection count is checked. If it's 0, the protection passes. Otherwise the DJNZ and such is run, which will wedge/crash the system.

Assuming the check passed (or the user hit reset) the next check is run:

```
 804F: 21 00 70 LD HL,7000 ;start of RAM
8052: DD 2E 00 LD IXL,0 ;detection count
8055: 01 00 04 LD BC,0400 ;1K of RAM
8058: 7E LD A,(HL) ;read RAM byte
8059: 23 INC HL
805A: FE 20 CP 20
805C: 38 0E JR C,806C ;jump is the RAM byte is <= 20 (this detects ASCII) 805E: FE
7B CP 7B 8060: 30 0A JR NC,806C ;jump if the RAM byte is > 7B
8062: DD 2C INC IXL ;increment detection count
8064: DD 7D LD A,IXL ;check IXL against 8
8066: FE 08 CP 08
8068: 28 E5 JR Z,804F ;if we detect 8 ASCII chars in a row, jump back to the start
of the detection code and wedge
806A: 18 03 JR 806F ;ASCII char, but we didn't find 8 yet
806C: DD 2E 00 LD IXL,0 ;reset count if we hit a non-ASCII character
806F: 0B DEC BC
8070: 78 LD A,B
8071: B1 OR A,C ;decrement and check byte count
8072: 20 E4 JR NZ,8058 ;do all bytes. fall through into the game code if it passes
```

This next check is interesting, because they use the lower 8 bits of IX as an 8 bit register. They are using "undocumented" opcodes for this. There is absolutely no reason, other than to obfuscate the code, or to cause issues with emulators that do not implement those opcodes.

The check here is fairly straight forward- it scans through the RAM and increments IXL if the value is in the range of 0x20-0x7B, which is the ASCII range. This range

occupies a bit less than 50% of the ‘byte space’, i.e. the values 0x00-0xFF. This means, there’s roughly a 1 in 512 chance that a particular Coleco can falsely trigger this check. The bad news is, RAM data on boot isn’t 100% random, and certain locations can have the same or very similar value each power on. It is determined by the small differences in each chip, and this means you might have an “unlucky” Coleco that always or mostly always falsely triggers it.

The fix is simple to bypass the protection, changing byte 0x000a in the ROM from 0x25 to 0x74 is it. This works because it changes the start address to 0x8074, which is the start of the game code.

Hope this was informative. It sure was fun to work all this stuff out.