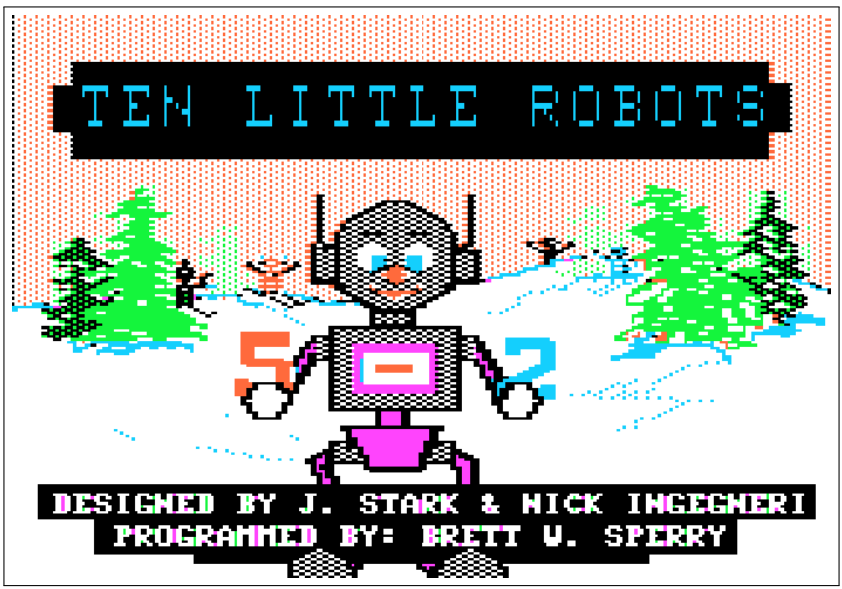
Ten Little Robots



2014-03-05

4am
to deprotect
and preserve

----------Ten Little Robots----------
A 4am crack 2014-03-05

Ten Little Robots is an educational game for young children that focuses on basic counting and math skills. I can not find any disk images of it anywhere online, so it's possible that this crack will be the first digital preservation of this game.

Copying with COPYA fails almost immediately with a disk read error. Copying with Disk Muncher 8 shows a read error on track $22; the copy boots, reads a few tracks, swings to the end of the disk, then reboots. Bit copying with EDD 4 gives no errors, but the copy exhibits the same behavior as the Disk Muncher copy. There is almost certainly a nibble count or other weirdness on track $22. Time for boot tracing...

S6D1 = Ten Little Robots original disk
S5D1 = my work disk

```
]CALL -151
*9600<C600.C6FFM
*96F8:AD E8 C0 4C 59 FF
*9600G
...
*2800<800.8FFM
*C500G
]BSAVE TLR T0S0,A$2800,L$100
```

Let's see what we have in T0S0.

$0801..$0816: set reset vector to reboot

$0817..$0819: push $04 to stack (suspicious)

$081A..$0834: initialize text screen, JSR to $08A0 which zaps hi-res page 1, show hi-res page 1

$0835..$083D: put $5C in zero page $3E, put RTS into $0801. This sets up a situation where the code can later JSR ($3E) (OK, that's not a real instruction, but you can fake it by calling a local subroutine that ends with an indirect jump) to read other sectors from track 0.

$083E..$0840: push $72 to stack (suspicious)

$0841..$0844: put $00 in zero page $FC (why?)

$0845..$084F: call subroutine at $0870 to read sectors 01-0A into $0900..$12FF

$0850..$0858: call subroutine at $0870 to read sectors 0B-0D into $9D00..$9FFF

$0859..$0869: loop to move $0400..$05FF to $0900..$0AFF (odd, since we just read data into $0900..$12FF and now we're overwriting part of it)

$086A..$086F: read sectors 0E-0F into $0400..$0500 (falls through to subroutine at $0870)

$0870..$0888: read sectors
  A = destination address (high byte)
  Y = first logical sector to read
  X = last logical sector to read

$0888: RTS

The final call to $0870 isn't actually a JSR; it just falls through from $086A..$086F. That means that there is no explicit jump to the next boot phase. But since two values were manually pushed to the stack, this RTS will "return" to address $0473.

So, to trap this, I'll need to change those manual stack pushes to set the "return" address to a callback routine under my control, then do some memory moves to capture the code stored in the text page.

```
96F8-   A9 97       LDA   #$97
96FA-   8D 18 08    STA   $0818
96FD-   A9 04       LDA   #$04
96FF-   8D 3F 08    STA   $083F
9702-   4C 01 08    JMP   $0801
9705-   A0 00       LDY   #$00
9707-   B9 00 04    LDA   $0400,Y
970A-   99 00 24    STA   $2400,Y
970D-   B9 00 05    LDA   $0500,Y
9710-   99 00 25    STA   $2500,Y
9713-   C8          INY
9714-   D0 F1       BNE   $9707
9716-   AD E8 C0    LDA   $C0E8
9719-   4C 59 FF    JMP   $FF59

*9600G
...
*3D00<9D00.9FFFM
*C500G
]BSAVE TLR 0400-05FF,A$2400,L$200
]BSAVE TLR 0900-12FF,A$900,L$A00
]BSAVE TLR 9D00-9FFF,A$3D00,L$300
```

The next phase of the bootloader starts at $0473 (before I so rudely interrupted it):

```
0473-   46 4A       LSR   $4A
0475-   20 A5 04    JSR   $04A5
0478-   A6 2B       LDX   $2B
047A-   8E E9 B7    STX   $B7E9
```

Wait a minute. Taking the slot number (x16) from $2B and storing it in $B7E9? That looks suspiciously like setting up a standard DOS 3.3 RWTS parameter table. But there was nothing loaded into the $B700 range yet. Which means that the subroutine at $04A5 must load the RWTS. Let's interrupt the boot after that JSR $04A5 and see what's up.

```
96F8-   A9 97       LDA   #$97
96FA-   8D 18 08    STA   $0818
96FD-   A9 04       LDA   #$04
96FF-   8D 3F 08    STA   $083F
9702-   4C 01 08    JMP   $0801
9705-   A9 60       LDA   #$60
9707-   8D 78 04    STA   $0478
970A-   20 73 04    JSR   $0473
970D-   AD E8 C0    LDA   $C0E8
9710-   4C 59 FF    JMP   $FF59
```

Just from a quick spot check of memory, it appears that the subroutine at $04A5 actually loads tracks 1 and 2 into $A000..$BFFF. Combined with the three sectors from track 0 in $9D00..$9FFF, that would be enough to make... an entire copy of DOS 3.3 (or some semblance thereof). Hmm.

Anyway, I'll save it to the work disk and sort it out later.

```
*2000<A000.BFFFM
*C500G
...
]BSAVE TLR A000-BFFF,A$2000,L$2000
```

Returning where I left off in the $0400 range, the bootloader continues with some more RWTS-related initialization, then moves an entire page from $0500 to $0200.

```
047D-   20 8E BE    JSR   $BE8E
0480-   A5 FC       LDA   $FC
0482-   99 78 04    STA   $0478,Y
0485-   4A          LSR
0486-   8D 78 04    STA   $0478
0489-   A0 00       LDY   #$00
048B-   B9 00 05    LDA   $0500,Y
048E-   99 00 02    STA   $0200,Y
0491-   88          DEY
0492-   D0 F7       BNE   $048B
```

It reads a single sector using the standard DOS 3.3 RWTS entry point...

```
0494-   A9 B7       LDA   #$B7
0496-   A0 E8       LDY   #$E8
0498-   20 B5 B7    JSR   $B7B5
```

Calls the relocated subroutine...

```
049B-   20 00 02    JSR   $0200
```

And finally pushes an address to the stack and "returns" to it.

```
049E-   A9 B7       LDA   #$B7
04A0-   48          PHA
04A1-   A9 01       LDA   #$01
04A3-   48          PHA
04A4-   60          RTS
```

I'd bet good money that the routine at $0200 (previously $0500) is a nibble check.

```
0500-   A9 0A       LDA   #$0A
0502-   85 2A       STA   $2A
0504-   AE E9 B7    LDX   $B7E9
0507-   BD 89 C0    LDA   $C089,X
050A-   BD 8E C0    LDA   $C08E,X
050D-   A9 AC       LDA   #$AC
050F-   85 48       STA   $48
0511-   A9 02       LDA   #$02
0513-   85 49       STA   $49
0515-   A9 80       LDA   #$80
0517-   85 2B       STA   $2B
0519-   C6 2B       DEC   $2B
051B-   F0 5B       BEQ   $0578
051D-   20 44 B9    JSR   $B944
0520-   B0 56       BCS   $0578
0522-   A5 2D       LDA   $2D
0524-   C9 0D       CMP   #$0D
0526-   D0 F1       BNE   $0519
0528-   A0 00       LDY   #$00
052A-   BD 8C C0    LDA   $C08C,X
052D-   10 FB       BPL   $052A
052F-   88          DEY
0530-   F0 46       BEQ   $0578
0532-   C9 D5       CMP   #$D5
0534-   D0 F4       BNE   $052A
0536-   A0 00       LDY   #$00
0538-   BD 8C C0    LDA   $C08C,X
053B-   10 FB       BPL   $0538
053D-   88          DEY
053E-   F0 38       BEQ   $0578
0540-   C9 E7       CMP   #$E7
0542-   D0 F4       BNE   $0538
0544-   BD 8C C0    LDA   $C08C,X
0547-   10 FB       BPL   $0544
                          [...]
```

```
0549-   C9 E7        CMP   ##$E7
054B-   D0 2B        BNE   $0578
054D-   BD 8C C0     LDA   $C08C,X
0550-   10 FB        BPL   $054D
0552-   C9 E7        CMP   ##$E7
0554-   D0 22        BNE   $0578
0556-   BD 8D C0     LDA   $C08D,X
0559-   A0 10        LDY   ##$10
055B-   24 06        BIT   $06
055D-   BD 8C C0     LDA   $C08C,X
0560-   10 FB        BPL   $055D
0562-   88           DEY
0563-   F0 13        BEQ   $0578
0565-   C9 EE        CMP   ##$EE
0567-   D0 F4        BNE   $055D
0569-   A0 07        LDY   ##$07
056B-   BD 8C C0     LDA   $C08C,X
056E-   10 FB        BPL   $056B
0570-   D1 48        CMP   ($48),Y
0572-   D0 04        BNE   $0578
0574-   88           DEY
0575-   10 F4        BPL   $056B
0577-   60           RTS
0578-   C6 2A        DEC   $2A
057A-   D0 99        BNE   $0515
```

Yep, that's a nibble check. If the check fails, execution falls through to $057C, a.k.a. The Badlands, which wipes all memory and reboots. There is no return code; the caller just assumes that if execution returns, everything is OK. So I can most likely just skip the check and continue.

```
; interrupt boot after T0S0
96F8-   A9 97       LDA   #$97
96FA-   8D 18 08    STA   $0818
96FD-   A9 04       LDA   #$04
96FF-   8D 3F 08    STA   $083F
9702-   4C 01 08    JMP   $0801
; disable nibble check
9705-   A9 60       LDA   #$60
9707-   8D 00 05    STA   $0500
; continue boot, no more interruptions
970A-   4C 73 04    JMP   $0473
```

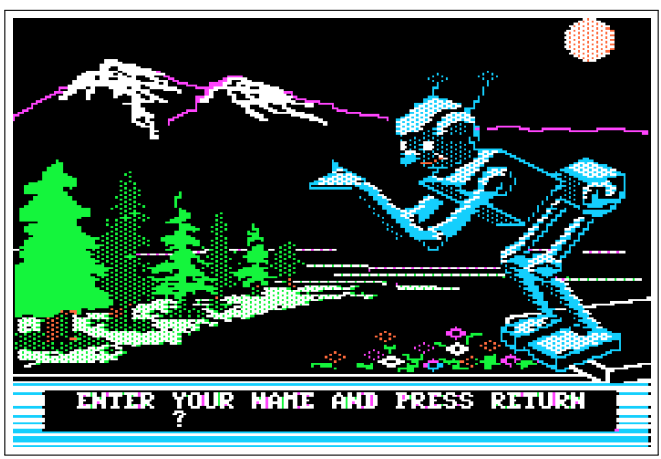
The game still boots; skipping the nibble check entirely appears to have no ill effects.

Now I think I have everything I need. A bit of spot checking confirms that the code that was loaded into $A000..$BFFF is indeed a full DOS 3.3. Not just an RWTS, but a full DOS. I know where the nibble check is and how to disable it. It's time to use Advanced Demuffin to make a standard DOS 3.3-readable disk.

```
]BLOAD TLR A000-BFFF,A$2000
]BSAVE TLR RWTS,A$2800,L$800

Put work disk in S6D1 and reboot

]PR#6
]BRUN ADVANCED DEMUFFIN 1.1

--> LOAD NEW RWTS MODULE
    At $B8, load "TLR RWTS" from D1

Put Ten Little Robots original disk in
S6D1, and a blank disk in S6D2.
```

```
--> FORMAT TARGET DISK

...grind grind grind...

--> CONVERT DISK

This disk is 16 sectors, and I want to
copy from track 03 sector 00 to track
21 sector 0F. (Track $22 is only used
for the nibble check, which I've
already bypassed.) The options screen
looks like this:


ADVANCED DEMUFFIN 1.1  - COPYRIGHT 1983
WRITTEN BY THE STACK -CORRUPT COMPUTING
========================================


INPUT ALL VALUES IN HEX


SECTORS PER TRACK? (13/16) 16

START TRACK: $03
START SECTOR: $00

END TRACK: $21
END SECTOR: $0F

INCREMENT: 1

MAX # OF RETRIES: 1

COPY FROM DRIVE 1
TO DRIVE: 2
========================================
16 SC $03,$00 TO $21,$0F BY $01 TO DRV2

Press RETURN to start the copy process:
```

```
ADVANCED DEMUFFIN 1.1  - COPYRIGHT 1983
WRITTEN BY THE STACK -CORRUPT COMPUTING
=======PRESS ANY KEY TO CONTINUE=======
TRK:  ................................
+.5:
   0123456789ABCDEF0123456789ABCDEF0123
SC0:  ................................
SC1:  ................................
SC2:  ................................
SC3:  ................................
SC4:  ................................
SC5:  ................................
SC6:  ................................
SC7:  ................................
SC8:  ................................
SC9:  ................................
SCA:  ................................
SCB:  ................................
SCC:  ................................
SCD:  ................................
SCE:  ................................
SCF:  ................................
========================================
16 SC $03,$00 TO $21,$0F BY $01 TO DRV2
```



At this point, I have the entire game on an unprotected disk, but no way to load it (since there's nothing on tracks 0-2 yet). But since I had confirmed that the original disk was loading a full DOS, I quit Advanced Demuffin and did a catalog:

```
]CATALOG,D2

C1983 DSR^C#254
012 FREE

 T 002 EXEC
 A 019 ADD
 A 015 MATCH
 A 014 COUNT
 A 049 STORY
 A 003 REW1
 A 002 REW2
 B 026 T3
 A 007 DRAW
 B 002 MUSIC
 B 008 GAME2
 B 034 ROBOT ADDITION
 B 002 LOMEM:
 B 034 ROBOT STORY
 B 034 ROBOT COUNT
 B 034 ROBOT FINAL2
 B 034 ROBO BYE BYE
 B 034 ROBO BYE BYE II
 A 004 REW3
 B 034 ROBOT ALPHA
 A 006 MENU
 B 005 TITLE.OBJ
 B 066 LOGO.PAGE
```

Well would you look at that! A regular DOS 3.3 disk catalog. There's no HELLO program, but the first file listed is a relatively short text file called EXEC.

```
]EXEC EXEC
```

Mirabile dictu! The entire game loads, title screen and all. There don't appear to be any further nibble checks or other secondary protection. The game even runs properly from drive 2! From this point on, all disk access is routed through normal DOS commands. In fact, most of the program is written in Applesoft BASIC.

But wait. If the game has a regular disk catalog and runs after booting from an entirely different disk, why do I need to recreate the obfuscated bootloader at all? I can just put a copy of DOS 3.3 on the unprotected disk and call it a day.

Well, I would still need to patch it to auto-exec the "EXEC" file instead of a HELLO program. Can DOS 3.3 do that? Beagle Bros. to the rescue! As listed on their inimitable "Peeks, Pokes & Pointers" chart, there is a magic POKE that will make DOS 3.3 auto-execute a text file instead of auto-running an Applesoft BASIC program.

- boot the DOS 3.3 master disk
- remove it and insert a blank disk

```
]POKE 40514,20
]INIT EXEC
```

I copied tracks $00-$02 from this scratch disk to the disk that I made with ADVANCED DEMUFFIN (with data on tracks $03-$21). I used Copy][+ for this because it has a convenient "Copy DOS" option, but Disk Muncher or Fast Copy or any other copier would also work.

Mirabile visu! The game loads and runs without a hitch. In a way, it seems anticlimactic. All that work, all that boot0 weirdness, code in the text pages, a custom RWTS, a nibble check, and for what? A lightly patched version of DOS 3.3 that runs a program written in Applesoft BASIC. And what about those sectors that were loaded from track 0 into $0900..$12FF? As far as I can tell, they were never used. Were they the remnants of another game that used a similar protection scheme? That was common practice in the 1980s. Perhaps I'm just stumbling over the ghosts of dead code, hastily reused and long forgotten.

.
.
.

Lastly, a postscript about a very interesting disk read routine used to read the rest of DOS from tracks $01 and $02...

At $0473, the bootloader called a subroutne at $040B. When I first traced the boot, I guessed that I could let this subroutine do its thing and regain control after it returned. I got lucky, and that guess turned out to be correct. But I want to delve into that subroutine a bit now, because it's interesting and it's something I haven't seen before.

This is the routine in its entirety:

```
0400-   A6 2B        LDX   $2B
0402-   6C 3E 00     JMP   ($003E)
0405-   20 0E 04     JSR   $040E
0408-   20 0E 04     JSR   $040E
040B-   20 0E 04     JSR   $040E
040E-   20 33 04     JSR   $0433
0411-   A2 0F        LDX   #$0F
0413-   A0 00        LDY   #$00
0415-   85 27        STA   $27
0417-   E8           INX
0418-   86 49        STX   $49
041A-   84 F9        STY   $F9
041C-   98           TYA
041D-   24 4A        BIT   $4A
041F-   30 03        BMI   $0424
0421-   B9 63 04     LDA   $0463,Y
0424-   85 3D        STA   $3D
0426-   20 00 04     JSR   $0400
0429-   A4 F9        LDY   $F9
042B-   C8           INY
042C-   C4 49        CPY   $49
042E-   90 EA        BCC   $041A
0430-   A5 27        LDA   $27
0432-   60           RTS
```

[...]

```
0433-   20 36 04    JSR   $0436
0436-   48          PHA
0437-   98          TYA
0438-   48          PHA
0439-   A5 FC       LDA   $FC
043B-   85 FD       STA   $FD
043D-   E6 FC       INC   $FC
043F-   A5 FC       LDA   $FC
0441-   29 03       AND   #$03
0443-   0A          ASL
0444-   05 2B       ORA   $2B
0446-   A8          TAY
0447-   B9 81 C0    LDA   $C081,Y
044A-   A9 30       LDA   #$30
044C-   20 A8 FC    JSR   $FCA8
044F-   A5 FD       LDA   $FD
0451-   29 03       AND   #$03
0453-   0A          ASL
0454-   05 2B       ORA   $2B
0456-   A8          TAY
0457-   B9 80 C0    LDA   $C080,Y
045A-   A9 30       LDA   #$30
045C-   20 A8 FC    JSR   $FCA8
045F-   68          PLA
0460-   A8          TAY
0461-   68          PLA
0462-   60          RTS
```

$0400..$0404: load X with the current slot number (x16) and jump to ($3E), which was set to point to $Cx5C way back in T0S0 ($0835..$083D). Furthermore, $0801 was set to an RTS, so you could JSR $0400 and it would end up returning control to you after reading a sector.

$0405..$040D: multiple calls to $040E, which appears to a main entry point of some sort. Note that the last one "falls through" to $040E (no RTS). So this is a cheap way of doing something multiple times. Calling $040E directly would do it once; calling $040B would do it twice; calling $0408 would do it three times; calling $0405 would do it 4 times. Nice. And it doesn't require an index register or any branching logic. The caller at $0473 called $040B, so this code will do it twice.

$040E..$0410: JSR $0433. Let's skip to that and then come back.

$0433..$0435: JSR $0436. Again, this "falls through" to $0436, so whatever $0436 is doing, this code will do it twice. I quite like this pattern, and apparently the original author did too.

$0436..$0438: push A and Y to the stack

$0439..$0446: load zero page $FC and manipulate it to... do what exactly? ORA with $2B? That's the current slot number (x16).

$0447..$045E: I'm afraid I'm not as familiar with the low-level disk motor control bits as I am with the higher level RWTS and DOS structure. So I went back to my dog-eared copy of "Beneath Apple DOS" and read through chapter 6 again ("Using DOS from Assembly Language"):

[begin quote]

ADDR	LABEL	DESCRIPTION
$C080	PHASE0FF	Step motor phase 0 off
$C081	PHASE0N	Step motor phase 0 on
$C082	PHASE1OFF	Step motor phase 1 off
$C083	PHASE1ON	Step motor phase 1 on
$C084	PHASE2OFF	Step motor phase 2 off
$C085	PHASE2ON	Step motor phase 2 on
$C086	PHASE3OFF	Step motor phase 3 off
$C087	PHASE3ON	Step motor phase 3 on

Basically, each of the four [stepper motor] phases (0-3) must be turned on and then off again. Done in ascending order, this moves the arm inward. In descending order, this moves the arm outward. The timing between accesses to these locations is critical, making this a non-trivial exercise.

[end quote]

Unsatisfied, I scoured the internet for some additional information to make sense of this. I found an archive of a single Usenet post from 1990 that explained how the stepper motors actually work.

macgui.com/usenet/?group=1&id=31160

[begin quote]

Basically, each track (and half-track) may be considered to be "under" one of the four phases of the stepper motor.

Track	Phase
0	0
0.5	1
1	2
1.5	3
2	0
2.5	1
3	2
3.5	3
etc.	

To figure the phase for a given (half-)track, multiply the track number by 2, and keep only the two low-order bits.

Stepping from one track to another is simply a matter of stepping one track at a time from the original track to the destination track. Thus, to step inward from track A to track B, first step to (half-)track A+0.5, then to (half-)track A+1, and so on, until you arrive at track B. Likewise, to step outward from track B to track A, first step to (half-)track B-0.5, then to B-1, and so on until you arrive at track A.

An individual step (which must from the original half-track to one if its immediately neighboring half-tracks) is accomplished by turning on the appropriate phase, waiting, and turning off the phase. An appropriate wait may be obtained by loading the accumulator with #$56 and doing a JSR to the Monitor's WAIT routine ($FCA8). (DOS and ProDOS are able to obtain improved speed by taking into account the fact that once the head is moving, it takes less time to make subsequent steps.)

Note that this scheme requires DOS to keep track of which track it's on--there's no way to ask the drive where the head is. If the current track number is unknown, the head must be "recalibrated" by assuming that we're currently at track 35 (or beyond), and then seeking to track 0 (this is what causes that awful GRRRRRINDing sound when you boot a 5.25" disk).

[end quote]

So, to seek from the current track to the next half track, you need to

1. Set up the Y register to be a slot number (x16) plus the appropriate phase (0-3, depending on which track the drive head is on)

2. LDA $C081,Y to turn on the appropriate stepper motor

3. Wait exactly the right amount of time (as measured in CPU cycles)

4. LDA $C080,Y to turn off the appropriate stepper motor

5. Wait the right amount of time again

...Which is exactly what this routine at $0436 is doing. And since $0433 "falls through" to $0436, it ends up doing this twice. Two half tracks equal one whole track, so calling the subroutine at $0433 will move the drive head to the next whole track. (By the way, this is why T0S0 initialized zero page $FC to $00 -- because that's the "current" track where the drive head is at boot.)

$0411..$0432: loop through from $0F down to $00 to read all the sectors on the current track and store them in the memory page whose high byte is passed in the accumulator (stored in zero page $27).

In case your eyes have glazed over by now (mine did repeatedly before I finally wrapped my head around all of this), let me give you a high-level summary: this clever routine uses some low-level drive magic and RE-USES CODE FROM THE 5.25 DRIVE CONTROLLER CARD IN ROM to read tracks $01-$02 into memory locations $A000..$BFFF.

If that were the end of the story, it would still be a good story, but it wouldn't have a whole lot to do with copy protection. Data read from disk; film at 11. But I couldn't figure out why those two tracks were so difficult to read. If I ran Copy][+ sector editor and set its option to use a patched DOS 3.3 RWTS, I was able to read all of track $00, and all of tracks $03-$21, but not track $01 or $02. If I ran Advanced Demuffin, I was able to use the original disk's own RWTS to read all of track $00, and all of tracks $03-$21, but not track $01 or $02. What the hell is going on with those tracks?

Finally, I broke out the Copy][+ nibble editor looked at the raw disk. Track $00 looks like this:

```
   COPY ][ PLUS BIT COPY PROGRAM 8.4
(C) 1982-9 CENTRAL POINT SOFTWARE, INC.
---------------------------------------

TRACK: 00  START: 324C  LENGTH: 015F

3228: FF FF FF FF FF FF FF FF   VIEW
3230: FF FF FF FF FF FF FF FF
3238: FF FF FF FF FF FF FF FF
3240: FF FF FF FF FF FF FF FF
3248: FF FF FF FF D5 AA 96 AA  <-324C
3250: AA AA AA AA AA AA AA FF
3258: FF FF FF 9F E7 F9 FE FF
3260: D5 AA AD D9 F2 B4 96 ED
3268: DF E7 D9 BF BE BE EE F7
```

The sequence "D5 AA 96" is the address prologue. The 8 bytes following that are AA AA AA AA AA AA AA AA. Decoding that, it says that this is track 0, sector 0 (true), and that this disk has volume number 0 (which is illegal). But the disk controller card doesn't actually check the volume number, so the disk still boots.

Looking at track $01, I noticed something VERY strange: the address field is exactly the same as track $00 (AA AA AA AA AA AA AA AA). Track $01 is claiming to be track $00. And track $02 is, too!

These tracks are lying to me.

No sane RWTS would be able to read these tracks. Any sane RWTS would barf on these tracks, because the track number listed in the address field doesn't match the track number it was trying to read. That's the entire purpose of the address field, so the RWTS can ensure it's reading data off the correct track and re-adjust the drive head if it's not.

But... these tracks aren't ever read by a sane RWTS. They're read by a very naive, very minimalist routine embedded in the disk controller card ROM. This routine doesn't do any checking of track numbers because it doesn't need to in order to fulfill its primary purpose (reading track $00). It has already slammed the drive head far enough that it can safely assume it's reading track $00, so it just blindly reads and never double-checks the track numbers in the address field. By manually moving the drive head, the original disk can re-use that naive routine in ROM to blindly read data from these intentionally malformed tracks. That's wickedly delicious.

A 4am crack 2014-03-05

----------EOF----------