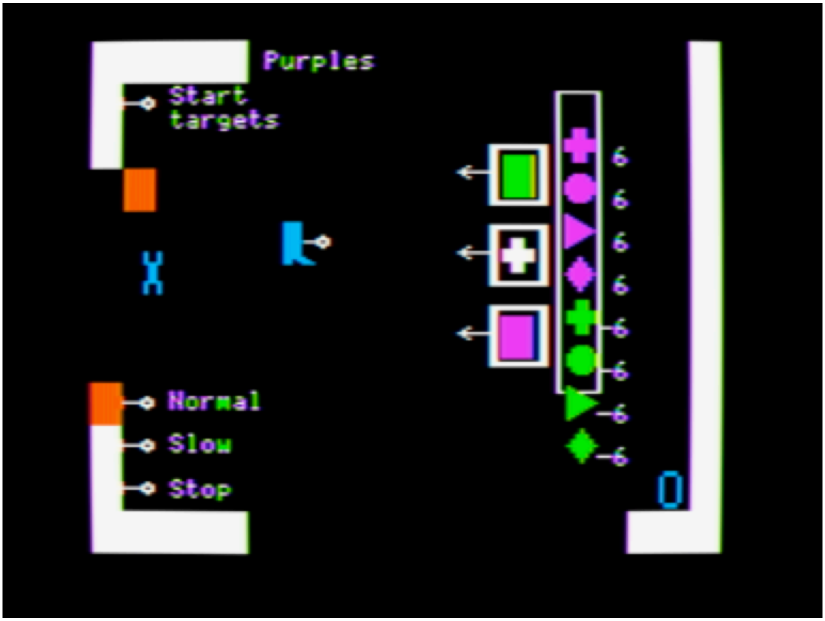
Rocky's Boots



2016-04-01

Contents

0 In Which Various Automated Tools Fail In Interesting Ways 4

1 In Which We Choose The Right Tool For The Job 7

2 In Which We Get Lucky 12

3 In Which Bits Never Lie 15

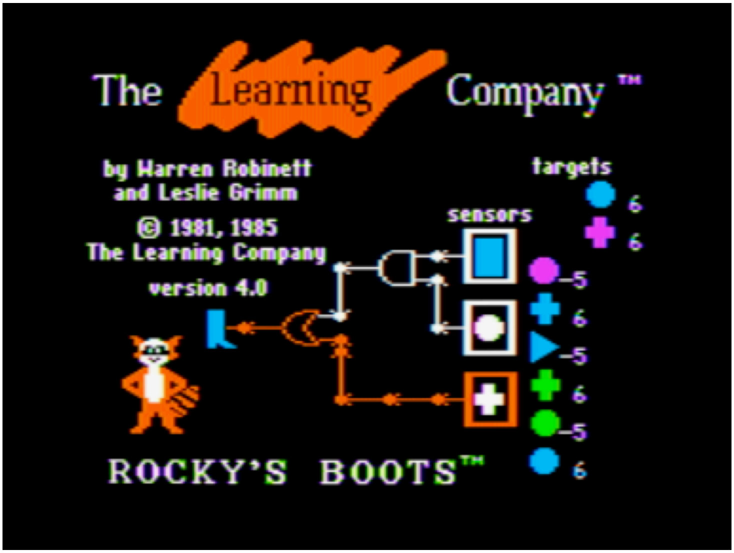
4 In Which We Take A Short Digression Into Some Theory So We Can Better Understand The Upcoming Bombshell 22

5 It's Not Just A Phase, Mom, This Is Who I Am 27

6 From Nibbles To Bytes And Back 32

A Acknowledgements 41

B Changelog 42



```
-------------Rocky's Boots--------------
A 4am crack                   2016-04-01
-------------------. updated 2016-04-12
                   |___________________
```

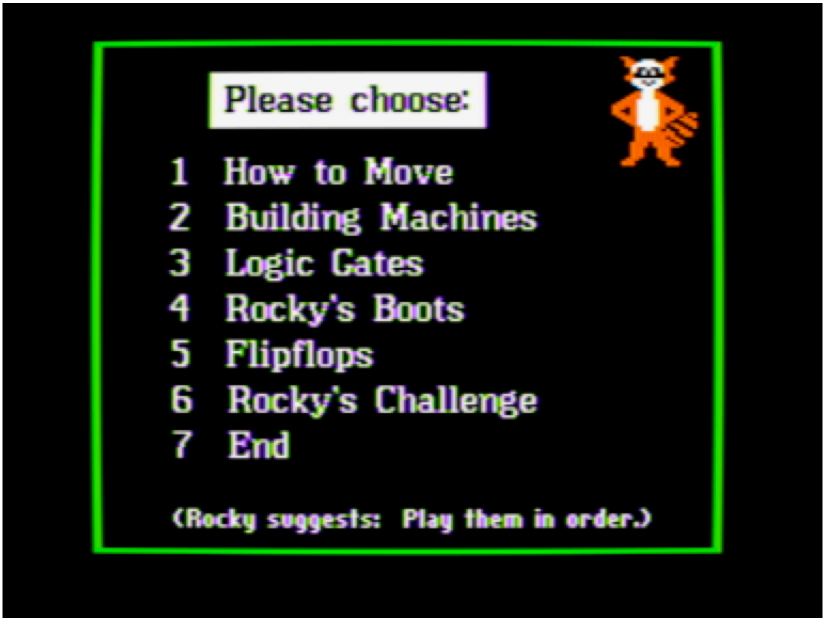
Name: Rocky's Boots
Version: 4.0
Genre: educational
Year: 1985
Authors: Warren Robinett and Leslie
  Grimm
Publisher: The Learning Company
Media: single-sided 5.25-inch floppy
OS: custom
Previous cracks: none of this version

Chapter 0
In Which Various Automated Tools Fail In Interesting Ways

```
COPYA
  immediate disk read error

Locksmith Fast Disk Backup
  unable to read any track

EDD 4 bit copy (no sync, no count)
  no errors, but copy swings to high
  track and reboots

Copy ][+ nibble editor
  all tracks use standard prologues
  (address: D5 AA 96, data: D5 AA AD)
  but modified epilogues
  (address: FF FF EB, data: FF FF EB)

Disk Fixer
  ["O" -> "Input/Output Control"]
    set Address Epilogue to "FF FF EB"
    set Data Epilogue to "FF FF EB"
  Success! All tracks readable!
  T00 -> custom bootloader
  no sign of DOS 3.3, ProDOS, or any
    kind of disk catalog

Why didn't COPYA work?
  modified epilogue bytes (every track)

Why didn't Locksmith FDB work?
  modified epilogue bytes (every track)

Why didn't my EDD copy work?
  probably a nibble check during boot
  (because disks do not spontaneously
  reboot unless someone tells them to)
```

Next steps:

1. Super Demuffin to convert the disk to a standard format
2. Patch the RWTS to read a standard disk (if necessary)
3. Find and disable the nibble check


Please choose:

1 How to Move
2 Building Machines
3 Logic Gates
4 Rocky's Boots
5 Flipflops
6 Rocky's Challenge
7 End

(Rocky suggests: Play them in order.)

Chapter 1
In Which We Choose
The Right Tool For The Job

I'm going to use Super Demuffin here
(instead of my usual go-to conversion
tool, Advanced Demuffin). The disk uses
a custom bootloader, so the AUTOTRACE
script on my work disk won't get very
far in capturing the RWTS. But luckily,
the RWTS modifications are minor --
custom epilogue bytes, same on every
track, and no apparent changes to the
nibble translation table -- so Super
Demuffin will work just fine.

When you first run Super Demuffin, it
asks for the parameters of the original
disk. In this case, the prologue bytes
are the same, but the epilogues are "FF
FF EB" instead of "DE AA EB".

```
      SUPER-DEMUFFIN AND FAST COPY
Modified by: The Saltine/Coast to Coast


   Address prologue: D5 AA 96

   Address epilogue: FF FF EB    DISK
                     ^^^^^     ORIGINAL
             *change from "DE AA"

      Data prologue: D5 AA AD

      Data epilogue: FF FF EB
                     ^^^^^
             *change from "DE AA"


 Ignore write errors while demuffining!


  D - Edit parameters
      <SPACE> - Advance to next parm
      <RETURN> - Exit edit mode
  R - Restore DOS 3.3 parameters
  O - Edit Original disk's parameters
  C - Edit Copy disk's parameters
  G - Begin demuffin process

                 --^--

Pressing "G" switches to the Locksmith
Fast Disk Copy UI. It assumes that both
disks are in slot 6, and that drive 1
is the original and drive 2 is the
copy.

[S6,D1=original disk]
[S6,D2=blank disk]
```

```
     LOCKSMITH 7.0  FAST DISK BACKUP


   R...................................
   W***********************************
HEX 00000000000000001111111111111111222
TRK 0123456789ABCDEF0123456789ABCDEF012
   0...................................
   1...................................
   2...................................
   3...................................
   4...................................
   5...................................
   6...................................
   7...................................
   8...................................
   9...................................
   A...................................
   B...................................
   C...................................
   D...................................
12 E...................................
   F...................................
[               ] PRESS [RESET] TO EXIT
```

--^--

There are two problems with this copy:

1. Depending on how the original disk was written, this copy may or may not be able to read itself. I may need to patch the disk's RWTS to deal with the fact that the disk is now in a standard format.

2. Even if it can read itself, it won't
   run. The copies I tried to make --
   even the bit copies -- just rebooted
   endlessly, which means there is some
   code being executed during boot to
   check if the disk is original.
   (Hint: it's not.)

Just by booting the copy, I can rule
out problem #1. The disk seems to read
itself just fine. It makes it exactly
as far as the failed bit copy -- far
enough to figure out that it's not an
original disk, and reboot.

Let's go find that protection check.

Chapter 2
In Which We Get Lucky

Since my copy reboots, and programs don't just do that without a good reason, I'm guessing there is a runtime protection check somewhere. One thing that all protection checks have in common is they need to turn on the drive motor by accessing a specific address in the $C0xx range. For slot 6, it's $C0E9, but to allow disks to boot from any slot, developers usually use code like this:

```
   LDX <slot number x 16>
   LDA $C089,X
```

There's nothing that says where the slot number has to be, although the disk controller ROM routine uses zero page $2B and lots of disks just reuse that. There's also nothing that says you have to use the X-register as the index, or that you must use the accumulator as the load register. But most RWTS code does, out of convention I suppose (or possibly fear of messing up such low-level code in subtle ways).

Also, since developers don't actually want people finding their protection-related code, they may try to encrypt it or obfuscate it to prevent people from finding it. But eventually, the code must exist and the code must run, and it must run on my machine, and I have the final say on what my machine does or does not do.

But sometimes you get lucky.

Turning to my trusty Disk Fixer sector
editor, I search the non-working copy
for "BD 89 C0", which is the opcode
sequence for "LDA $C089,X".

```
[Disk Fixer]
  ["F"ind]
    ["H"ex]
      ["BD 89 C0"]

                 --v--

------------- DISK SEARCH -------------

$00/$01-$1B   $00/$0C-$4F   $00/$0F-$52

                 --^--
```

Looking at T00,S01, it's immediately
obvious that I've hit the jackpot.

Chapter 3
In Which Bits Never Lie

Here is T00,S01 from the beginning, as seen through Disk Fixer's built-in disassembler. This sector is loaded into memory at $4F00. (Oddly enough, T00,S00 is almost identical to standard the DOS 3.3 boot0, except it loads all of track $00 into $4E00+.) This is the first thing called after boot0 finishes reading track $00.

--v--

```
T00,S01
------------ DISASSEMBLY MODE ----------
; save zero page
0000:A0 FF              LDY   #$FF
0002:B9 00 00           LDA   $0000,Y
0005:99 00 5E           STA   $5E00,Y
0008:88                 DEY
0009:D0 F7              BNE   $0002

; seek to track $22 (not shown)
000B:A9 00              LDA   #$00
000D:8D 78 04           STA   $0478
0010:A9 44              LDA   #$44
0012:20 A0 56           JSR   $56A0

; high byte of Death Counter
0015:A9 0A              LDA   #$0A
0017:85 F0              STA   $F0

; turn on drive motor of the slot we
; just booted from (stored in zp$2B)
0019:A6 2B              LDX   $2B
001B:BD 89 C0           LDA   $C089,X
001E:BD 8E C0           LDA   $C08E,X
```

```
; probably an address, ($F2) -> $4FD8
0021:A9 D8             LDA   #$D8
0023:85 F2             STA   $F2
0025:A9 4F             LDA   #$4F
0027:85 F3             STA   $F3

; low byte of Death Counter
0029:A9 80             LDA   #$80
002B:85 F1             STA   $F1
002D:C6 F1             DEC   $F1

; when Death Counter hits 0, bad things
; happen
002F:F0 5C             BEQ   $008D

; this finds the next address prologue
; ("D5 AA 96") and skips over the
; address field
0031:20 44 56          JSR   $5644

; if that didn't work, fail
0034:B0 57             BCS   $008D

; loop until we find sector $0F (in
; zero page $2D after routine at $5644)
0036:A5 2D             LDA   $2D
0038:C9 0F             CMP   #$0F
003A:D0 F1             BNE   $002D

; here we go
003C:A0 00             LDY   #$00
003E:BD 8C C0          LDA   $C08C,X
0041:10 FB             BPL   $003E
0043:88                DEY
0044:F0 47             BEQ   $008D  ; fail

; find $D5 nibble
0046:C9 D5             CMP   #$D5
0048:D0 F4             BNE   $003E
```

```
; find $E7 $E7 $E7 sequence of nibbles
; within the next $100 nibbles
; (Y register serves as the mini-Death
; Counter here, if it wraps around to 0
; then we give up)
004A:A0 00              LDY   #$00
004C:BD 8C C0           LDA   $C08C,X
004F:10 FB              BPL   $004C
0051:88                 DEY
0052:F0 39              BEQ   $008D   ; fail
0054:C9 E7              CMP   #$E7
0056:D0 F4              BNE   $004C
0058:BD 8C C0           LDA   $C08C,X
005B:10 FB              BPL   $0058
005D:C9 E7              CMP   #$E7
005F:D0 2C              BNE   $008D   ; fail
0061:BD 8C C0           LDA   $C08C,X
0064:10 FB              BPL   $0061
0066:C9 E7              CMP   #$E7
0068:D0 23              BNE   $008D   ; fail

; reset data latch and kill some time
; to get out of sync with the original
; nibble boundary
006A:BD 8D C0           LDA   $C08D,X

; reset Y (serves as a mini-Death
; Counter again in the next loop)
006D:A0 10              LDY   #$10

; does nothing of consequence except
; burn a few more CPU cycles
006F:24 06              BIT   $06
```

```
; now start looking for nibbles that
; don't really exist (except they do,
; because we're out of sync and reading
; timing bits as data)
0071:BD 8C C0          LDA   $C08C,X
0074:10 FB             BPL   $0071
0076:88                DEY
0077:F0 14             BEQ   $008D  ; fail
0079:C9 EE             CMP   #$EE
007B:D0 F4             BNE   $0071

; check for (still desynchronized)
; nibble sequence stored in reverse
; order at ($F2) a.k.a. $4FD8
007D:A0 07             LDY   #$07
007F:BD 8C C0          LDA   $C08C,X
0082:10 FB             BPL   $007F
0084:D1 F2             CMP   ($F2),Y
0086:D0 05             BNE   $008D  ; fail
0088:88                DEY
0089:10 F4             BPL   $007F
008B:30 03             BMI   $0090  ; pass

; failure path ends up here, from
; multiple places noted above --
; unconditionally branch further down
008D:18                CLC
008E:90 3E             BCC   $00CE

; successful execution continues here
; (from $4F8B)
0090:A9 60             LDA   #$60
0092:8D 01 08          STA   $0801
```

```
; move the rest of this page to higher
; memory
0095:A0 80              LDY   #$80
0097:B9 00 4F           LDA   $4F00,Y
009A:99 00 5F           STA   $5F00,Y
009D:C8                 INY
009E:D0 F7              BNE   $0097

; and continue there
00A0:4C A3 5F           JMP   $5FA3

; set up zero page so we can call the
; disk controller ROM to read a sector
; (from track $22, which we're still
; on after the successful protection
; check above)
; sector $02
00A3:A9 02              LDA   #$02
00A5:85 3D              STA   $3D

; track $22
00A7:A9 22              LDA   #$22
00A9:85 41              STA   $41

; address $4F00
00AB:A9 4F              LDA   #$4F
00AD:85 27              STA   $27
00AF:A9 00              LDA   #$00
00B1:85 26              STA   $26
00B3:A5 3F              LDA   $3F
00B5:8D BA 5F           STA   $5FBA

; read it
00B8:20 5C C6           JSR   $C65C

; seek back to track $00
00BB:A9 00              LDA   #$00
00BD:20 A0 56           JSR   $56A0
```

```
; restore zero page
00C0:A0 00          LDY   #$00
00C2:B9 00 5E       LDA   $5E00,Y
00C5:99 00 00       STA   $0000,Y
00C8:88             DEY
00C9:D0 F7          BNE   $00C2

; and jump to actual boot1 code, which
; we just read from track $22
00CB:4C 00 4F       JMP   $4F00

; failure path continues here (from the
; unconditional branch at $4F8E) --
; decrement the high byte of the Death
; Counter and either jump back to try
; again or give up and reboot
00CE:C6 F0          DEC   $F0
00D0:D0 03          BNE   $00D5
00D2:4C 00 C6       JMP   $C600
00D5:4C 29 4F       JMP   $4F29

; array nibbles to look for after we've
; intentionally desynchronized (by
; burning CPU cycles at $4F6A..$4F6F)
00D8:FC EE EE FC E7 EE FC E7
```

--^--

We can bypass this entire protection
check by forcing it down the success
path (at $4F90). But that's not what
we're going to do.

We're going to do one better.

Chapter 4
In Which We Take A Short Digression Into Some Theory So We Can Better Understand The Upcoming Bombshell

$E7 $E7 $E7 $E7. What would that nibble sequence look like on disk? The answer is, "It depends." $E7 in hexadecimal is 11100111 in binary, so here is the simplest possible answer:

```
   |--E7--||--E7--||--E7--||--E7--|
   11100111111001111110011111100111
```

But wait. Every nibble read from disk must have its high bit set. In theory, you could insert one or two "0" bits after any of those nibbles. (Two is the maximum, due to hardware limitations.) These extra "0" bits would be swallowed by the standard "wait for data latch to have its high bit set" loop, which you see over and over in any RWTS code:

```
  :1   LDA $C08C,X
       BPL :1
```

Now consider the following bitstream:

```
  |--E7--| |--E7--|  |--E7--||--E7--|
  11100111011100111001110011111100111
          ^        ^^
       (extra)   (extra)
```

The first $E7 has one extra "0" bit after it, and the second $E7 has two extra "0" bits after it. Totally legal, works on any Apple II computer and any floppy drive. A "LDA $C08C,X; BPL" loop would still interpret this bitstream as a sequence of four $E7 nibbles. Each of the extra "0" bits appear after we've just read a nibble and we're waiting for the high bit to be set again. They get "swallowed." Ignored. Like they were never there.

But what if we miss the first few bits of this bitstream, then start looking? The disk is always spinning, whether we're reading from it or not. If we waste too much time doing something other than reading, we'll literally miss some bits as the disk spins. (This is why the timing of low-level RWTS code is so critical.)

Let's say we waste 12 CPU cycles before we start reading this bitstream. Each bit takes 4 CPU cycles to go by, so after 12 cycles, we would have missed the first 3 bits (marked with an X).

```
            (normal start)

  |--E7--| |--E7--|  |--E7--||--E7--|
  11100111011100111001110011111100111
  XXX  |--EE--| |--E7--|  |--FC--|

           (delayed start)
```

Ah! It's interpreted as a completely
different nibble sequence if you delay
just a few CPU cycles before you start
reading. Also note that some of those
"extra" bits are no longer being
ignored; now they're being interpreted
as data, as part of the nibbles that
are being returned to the higher level
code. Meanwhile, other bits that were
part of the $E7 nibbles are now being
swallowed.

Now, let's go back to the first stream,
which had no extra bits between the
nibbles, and see what happens when we
waste those same 12 CPU cycles.

```
           (normal start)

   |--E7--||--E7--||--E7--||--E7--|
   11100111111001111110011111100111
   XXX  |--FC--||--FC--||--FC--|

          (delayed start)
```

After skipping the first three bits,
the stream is interpreted as a series
of $FC $FC $FC repeating endlessly --
not $EE $E7 $FC like the other stream.

Here's the kicker: generic bit copiers didn't preserve these extra "0" bits between nibbles. Even top-of-the-line bit copiers couldn't reliably detect the difference between 1 timing bit and 2 timing bits. By "desynchronizing" (wasting just the right number of CPU cycles at just the right time), then interpreting the bits on the disk in mid-stream, developers could determine at runtime whether you had an original disk.

Here is the complete "E7 bitstream," annotated to show both the synchronized and desynchronized nibble sequences.

```
 |--E7--| |--E7--|  |--E7--||--E7--|
 111001110111001110011100111111001110
 XXX  |--EE--| |--E7--|  |--FC--||--E

 |--E7--|  |--E7--||--E7--| |--E7--|
 111001110011100111111001110111001110
 E--| |--E7--|  |--FC--||--EE--| |--E

 |--E7--||--E7--|
 1110011111100111
 E--| |--FC--|
```

Chapter 5
It's Not Just A Phase, Mom, This Is Who I Am

This protection scheme hinges on the assumption that only an original disk will present the proper sequence of nibbles after the code intentionally "desynchronizes" mid-stream. This seems like an entirely reasonable assumption. After all, even the best bit copiers could not preserve the exact number of timing bits after every single nibble.

However, that assumption rests on a deeper assumption. Once it burns 12 CPU cycles (skipping 3 bits and getting out of sync with the original nibble boundary), it assumes that the next nibbles it reads (EE E7 FC EE E7 FC EE EE FC) are dependent on the timing bits that were originally between the $E7 nibbles.

In other words, it assumes that once it gets out of sync, it stays out of sync.

But what if that weren't true?

What if we could resynchronize the bitstream to the original nibble boundary -- after the code burned time intentionally to get out of sync? Imagine a sequence of nibbles which, when read by this code, would swallow an additional 5 bits and get back in sync with the original nibble boundary.

This would need to happen before the
code found the $EE nibble, because
immediately after that, it starts
reading additional nibbles and checking
them against a hard-coded array. But
there is a small window there, after we
desynchronize but before we find the
$EE nibble. Here is the relevant code:

```
; burn CPU cycles to get out of sync
006A:BD 8D C0         LDA   $C08D,X
006D:A0 10            LDY   #$10
006F:24 06            BIT   $06

; now start looking for an $EE nibble
0071:BD 8C C0         LDA   $C08C,X
0074:10 FB            BPL   $0071
0076:88               DEY
0077:F0 14            BEQ   $008D  ; fail
0079:C9 EE            CMP   #$EE
007B:D0 F4            BNE   $0071
```

The Y register gets reset to $10 (at
$4F6D) and is decremented while we're
looking for the $EE nibble. That means
we can skip up to 15 nibbles between
the third $E7 and $EE, and execution
will still continue without branching
to the failure path.

Now watch this:

```
           (normal start)

   |--EF--||--F3--||--FC--||--EE--|
   11101111111100111111110011101110
   XXX |--FF--|  |--FF--|  |--EE--|

          (delayed start)
```

If we put this bitstream immediately after the initial $E7 $E7 $E7 sequence, the loop at $4F71..$4F7C that's looking for the desynchronized $EE nibble will instead find two desynchronized $FF nibbles, skip over them (decrementing the Y register, but not all the way to zero), then finally find the $EE nibble and be happy. But look what happened in the meantime: we've resynchronized the bitstream to the original nibble boundary! By putting "0" bits before and after each desynchronized $FF nibbles, we've essentially "swallowed" the extra bits and shifted the phase from +3 to +8. Since nibbles are 8-bit values, a phase of +8 is the same as 0.

In just 24 bits, we've resynchronized the bitstream and fooled the protection check into reading regular nibbles as if they were desynchronized nibbles. If we put this nibble sequence in the right place on our non-working copy, we don't need to bypass the protection code at all. The code can run as usual. Let it run. Let it search for nibbles. Let it desynchronize. Let it verify the magic nibble stream. It'll all pass.

We've defeated the E7 protection check with no code changes.

Chapter 6
From Nibbles To Bytes And Back

The E7 bitstream is part of the data
field of a real sector on disk. On this
disk, it's on T22,S0F. (We saw the code
seek to track $22 at $4F10 and seek to
sector $0F at $4F31.) Here's what it
looks like in the Copy][+ nibble
editor:

```
                 --v--

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

TRACK: 22  START: 1800  LENGTH: 3DFF

1AF0: FF FF FF FF FF D5 AA 96   VIEW
                     ^^^^^^^^
                 address prologue

1AF8: FF FE BB AA AF AF EB FB
      ^^^^^ ^^^^^ ^^^^^ ^^^^^
      V=254 T=$22 S=$0F chksm

1B00: FF FF EB 9B FC FF FF FF
      ^^^^^^^^
  address epilogue

1B08: FF FF FF FF FF FF FF D5
                           ^^

1B10: AA AD 96 96 96 96 96 96
      ^^^^^
  data prologue
```

```
1B18:  96  96  96  96  96  96  96  96
1B20:  96  96  96  96  96  96  96  96
1B28:  96  96  96  96  96  96  96  96
1B30:  96  96  96  96  96  96  96  96
1B38:  96  96  96  96  96  96  96  96
1B40:  96  96  96  96  96  96  96  96
1B48:  96  96  96  96  96  96  96  96
1B50:  96  96  96  96  96  96  96  96
1B58:  96  96  96  96  96  96  96  96
1B60:  96  96  96  96  96  96  96  96
1B68:  96  96  96  96  96  96  96  96
1B70:  96  96  96  96  96  96  96  96
1B78:  96  96  96  96  96  96  96  96
1B80:  96  96  96  96  96  96  96  96
1B88:  96  96  96  96  96  96  96  96
1B90:  96  96  96  96  96  96  96  96
1B98:  96  96  96  96  96  96  96  96
1BA0:  96  96  96  96  96  96  96  96
1BA8:  96  96  96  96  96  96  96  96
1BB0:  96  96  96  96  96  96  96  96
1BB8:  96  96  96  96  96  96  96  96
1BC0:  96  96  96  96  96  96  96  96
1BC8:  96  96  96  96  96  96  96  96
1BD0:  96  96  96  96  96  96  96  96
1BD8:  96  96  96  96  96  96  96  96
1BE0:  96  96  96  96  96  96  96  96
1BE8:  96  96  96  96  96  96  96  96
1BF0:  96  96  96  96  96  96  96  96
1BF8:  96  96  96  96  96  96  96  96
[...]
```

```
1C00: 96 96 96 96 96 96 96 96
1C08: E7 E7 E7 E7 E7 E7 E7 E7   ; timing
1C10: E7 E7 E7 E7 E7 E7 E7 E7   ; bits
1C18: E7 E7 E7 E7 E7 E7 E7 E7   ; are in
1C20: E7 E7 E7 E7 E7 E7 E7 E7   ; here
1C28: E7 E7 E7 E7 E7 E7 E7 E7
1C30: E7 E7 E7 E7 E7 E7 E7 E7
1C38: E7 E7 E7 E7 E7 E7 E7 E7
1C40: E7 E7 E7 E7 E7 E7 E7 E7
1C48: E7 E7 E7 E7 E7 E7 E7 E7
1C50: E7 E7 E7 E7 E7 E7 E7 E7
1C58: E7 E7 E7 E7 E7 E7 E7 E7
1C60: E7 E7 E7 E7 E7 E7 E7 E7
1C68: 96 FF FF EB F7 F9 FE FF
         ^^^^^^^^
      data epilogue

---------------------------------------

  A  TO ANALYZE DATA  ESC TO QUIT

  ?  FOR HELP SCREEN  /  CHANGE PARMS

  Q  FOR NEXT TRACK   SPACE TO RE-READ

                 --^--
```

Remember how the protection check looks for a $D5 nibble (starting at $4F3C)? The one it finds is the first nibble of the data prologue, shown here at offset $1B0F.

Remember how the protection check gives
itself a $100 nibble window to find the
first $E7 nibble after it finds a $D5?
It's skipping over most of the data
field -- a bunch of $96 nibbles.

This is a real sector. I mean, the code
never reads T22,S0F as sector data, but
you can see it in your favorite sector
editor if you want. It looks like this:

```
                 --v--

-------------- DISK EDIT --------------
TRACK $22/SECTOR $0F/VOLUME $FE/BYTE$00
---------------------------------------
$00: 00 00 00 00 00 00 00 00   ........
$08: 00 00 00 00 00 00 00 00   ........
$10: 00 00 00 00 00 00 00 00   ........
$18: 00 00 00 00 00 00 00 00   ........
$20: 00 00 00 00 00 00 00 00   ........
$28: 00 00 00 00 00 00 00 00   ........
$30: 00 00 00 00 00 00 00 00   ........
$38: 00 00 00 00 00 00 00 00   ........
$40: 00 00 00 00 00 00 00 00   ........
$48: 00 00 00 00 00 00 00 00   ........
$50: 00 00 00 00 00 00 00 00   ........
$58: 00 00 00 00 00 00 00 00   ........
$60: 00 00 00 00 00 00 00 00   ........
$68: 00 00 00 00 00 00 00 00   ........
$70: 00 00 00 00 00 00 00 00   ........
$78: 00 00 00 00 00 00 00 00   ........
$80: 00 00 00 00 00 00 00 00   ........
$88: 00 00 00 00 00 00 00 00   ........
$90: 00 00 00 00 00 00 00 00   ........
[...]
```

```
$98: 00 00 00 00 00 00 00 00   ........
$A0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$A8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$B0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$B8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$C0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$C8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$D0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$D8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$E0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$E8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$F0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
$F8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
---------------------------------------
BUFFER 0/SLOT 6/DRIVE 1/MASK OFF/NORMAL

---------------------------------------
COMMAND : _
```

--^--

Those $00 bytes in memory are written
to disk as $96 nibbles. No magic here,
just the standard 6-and-2 encoding that
DOS 3.3 uses to convert bytes in memory
to nibbles on disk. Furthermore, that
repeated pattern of "AC 00", starting
at offset $A0, are the $E7 nibbles at
the end of the data field (shown above
in the Copy][+ nibble ditor at offset
$1C08).

My non-working copy looks identical to
this, except it lacks the timing bits
between the $E7 nibbles, so the
protection check fails. But let's make
a 12-byte patch:

```
T22,S0F,$A3
- "00 AC 00 AC 00 AC 00 AC 00 AC 00 AC"
+ "64 B4 44 80 2C DC 18 B4 44 80 44 B4"
```

Now the sector looks like this:

```
                 --v--

------------- DISK EDIT ---------------
TRACK $22/SECTOR $0F/VOLUME $FE/BYTE$00
---------------------------------------
$00: 00 00 00 00 00 00 00 00   ........
.
. [unchanged]
.
$98: 00 00 00 00 00 00 00 00   ........
$A0: AC 00 AC 64 B4 44 80 2C   ,.,d4D.,
$A8: DC 18 B4 44 80 44 B4 00   \.4D.D4.
$B0: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
.
. [unchanged]
.
$F8: AC 00 AC 00 AC 00 AC 00   ,.,.,.,.
---------------------------------------
BUFFER 0/SLOT 6/DRIVE 1/MASK OFF/NORMAL

---------------------------------------
COMMAND : _

                 --^--
```

Moving back to the Copy][+ nibble
editor, the sector looks like this:

```
                 --v--

TRACK: 22  START: 1800  LENGTH: 3DFF

1AF0: FF FF FF FF FF D5 AA 96   VIEW
1AF8: FF FE BB AA AF AF EB FB
1B00: FF FF EB 9B FC FF FF FF
1B08: FF FF FF FF FF FF FF D5
1B10: AA AD 96 96 96 96 96 96
.
. [unchanged]
.
1C00: 96 96 96 96 96 96 96 96
1C08: E7 E7 E7 EF F3 FC EE E7   ; no
1C10: FC EE E7 FC EE EE FC EA   ; timing
1C18: E7 E7 E7 E7 E7 E7 E7 E7   ; bits
1C20: E7 E7 E7 E7 E7 E7 E7 E7   ; here!
.
. [unchanged]
.
1C68: 96 FF FF EB F7 F9 FE FF

                 --^--
```

When the protection check looks for a
$D5 nibble, it will find it. (It's the
first nibble of the data prologue; that
hasn't changed.)

When the protection check looks for an
$E7 nibble, it will find it. (It's at
offset $1C08, after $F6 other nibbles
that are all $96. That hasn't changed
either.)

When it desynchronizes and looks for an $EE nibble, it will find it (after it skips over two desynchronized $FF nibbles) at offset $1C0E. At this point we've resynchronized the bitstream to the original nibble boundary, but the protection code doesn't know that.

The next 8 nibbles on disk after $EE (starting at offset $1C0F) are "E7 FC EE E7 FC EE EE FC". That's the magic nibble sequence that the protection check looks for. It thinks it's reading out-of-phase nibbles from an original disk, but it's really reading in-sync nibbles from a specially crafted copy.

```
]PR#6
```
...works, and it is glorious...

Quod erat liberandum.

Acknowledgements

Many, many thanks to qkumba for doing all of the heavy lifting here. This entire technique is his idea, and he spent a lot of time getting it to work in modern emulators (with their, um, "special" ways of reading from emulated disk images). I spent a comparatively small amount of time minimizing the byte patch. The final result may look simple, but there were a lot of almost-but-not-quite solutions along the way. Read all about it here:

http://www.alchemistowl.org/pocorgtfo/pocorgtfo11.pdf

Changelog

2016-04-12

- fixed date on the previous update, because I am not a time traveler (thanks Ange)

2016-04-05

- typos (thanks Ange)