WAVY NAVY

2016-04-21

4am
to deprotect and preserve

Contents

SIRIUS PRESENTS
WAVY
NAVY
BY
RODNEY MC AULEY
1 3
0000
0
0000
0
0000
0
0000
HIGH
SCORE
0000

```
---------------Wavy Navy----------------
A 4am crack                   2016-04-21
----------------------------------------


Name: Wavy Navy
Genre: arcade
Year: 1982
Authors: Rodney McAuley
Publisher: Sirius Software
Media: single-sided 5.25-inch floppy
OS: custom
Previous cracks: The Cloak, LoGo
Similar cracks:
  #392 Flip Out
```

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)
  loads entire game, then prints
  "SIRIUS" and reboots

Copy][+ nibble editor
  lower tracks have what appears to be
  4-4 encoded data

Disk Fixer
  nope (can't read 4-4 encoded tracks)

Why didn't COPYA work?
  not a 16-sector disk

Why didn't Locksmith FDB work?
  ditto

Why didn't my EDD copy work?
  Presumably a protection check before
  the game starts. Disks do not reboot
  unless someone tells them to.

This is a single-load game. It never
accesses the disk once it's loaded into
memory.

Next steps:

  1. Trace the boot
  2. Capture the game in memory
  3. Write it out to a standard disk
     with some kind of fastloader

Chapter 1
In Which We Start Off Loudly And Build To A Crescendo

```
[S6,D1=original disk]
[S5,D1=my work disk]

]PR#5
CAPTURING BOOT0
...reboots slot 6...
...reboots slot 5...
SAVING BOOT0

]BLOAD BOOT0,A$800
]CALL -151

*801L

; save boot slot number
0801-   A5 2B         LDA   $2B
0803-   AA            TAX
0804-   85 FB         STA   $FB
0806-   4A            LSR
0807-   4A            LSR
0808-   4A            LSR
0809-   4A            LSR
080A-   09 C0         ORA   #$C0
080C-   8D 00 30      STA   $3000

; zap language card RAM bank
080F-   A0 00         LDY   #$00
0811-   84 00         STY   $00
0813-   A9 D0         LDA   #$D0
0815-   85 01         STA   $01
0817-   A2 30         LDX   #$30
0819-   AD 81 C0      LDA   $C081
081C-   AD 81 C0      LDA   $C081
081F-   B1 00         LDA   ($00),Y
0821-   91 00         STA   ($00),Y
0823-   C8            INY
0824-   D0 F9         BNE   $081F
0826-   E6 01         INC   $01
0828-   CA            DEX
0829-   D0 F4         BNE   $081F
```

```
; initialize globals
082B-   A6 FB       LDX   $FB
082D-   84 F7       STY   $F7
082F-   A9 BB       LDA   #$BB
0831-   85 F8       STA   $F8
0833-   A9 04       LDA   #$04
0835-   85 FA       STA   $FA

; load some more sectors from track $00
; with a 4-4 encoding scheme and a
; prologue of "AD DA DD"
0837-   BD 8C C0    LDA   $C08C,X
083A-   10 FB       BPL   $0837
083C-   C9 AD       CMP   #$AD
083E-   D0 F7       BNE   $0837
0840-   BD 8C C0    LDA   $C08C,X
0843-   10 FB       BPL   $0840
0845-   C9 DA       CMP   #$DA
0847-   D0 F3       BNE   $083C
0849-   BD 8C C0    LDA   $C08C,X
084C-   10 FB       BPL   $0849
084E-   C9 DD       CMP   #$DD
0850-   D0 EA       BNE   $083C
0852-   A0 00       LDY   #$00
0854-   84 F5       STY   $F5

; main loop to read 2 nibbles and save
; 1 byte
0856-   BD 8C C0    LDA   $C08C,X
0859-   10 FB       BPL   $0856
085B-   38          SEC
085C-   2A          ROL
085D-   85 F6       STA   $F6
085F-   B0 11       BCS   $0872
0861-   BD 8C C0    LDA   $C08C,X
0864-   10 FB       BPL   $0861
0866-   2A          ROL
0867-   85 F6       STA   $F6
0869-   C8          INY
086A-   D0 06       BNE   $0872
```

```
; increment page
086C-   E6 F8          INC   $F8

; decrement sector count
086E-   C6 FA          DEC   $FA
0870-   F0 0F          BEQ   $0881

; calculate a running checksum
0872-   BD 8C C0       LDA   $C08C,X
0875-   10 FB          BPL   $0872
0877-   25 F6          AND   $F6
0879-   91 F7          STA   ($F7),Y
087B-   45 F5          EOR   $F5
087D-   85 F5          STA   $F5

; loop back for more bytes
087F-   B0 E0          BCS   $0861

; verify checksum
0881-   BD 8C C0       LDA   $C08C,X
0884-   10 FB          BPL   $0881
0886-   25 F6          AND   $F6
0888-   45 F5          EOR   $F5
088A-   D0 A3          BNE   $082F

; jump to the code we just loaded
088C-   4C 29 BB       JMP   $BB29
```

$F8 (initially $BB) appears to be the page in memory to put the sector. It's incremented after each read (at $086C).

$FA (initially $04) appears to be the sector count. It's decremented after each read (at $086E).

At $088C, it jumps to $BB29 to continue with the next phase of the boot. That's where I need to patch it.

```
*9600<C600.C6FFM

; set up callback to my code after RWTS
; is loaded into $BB00
96F8-   A9 05       LDA   #$05
96FA-   8D 8D 08    STA   $088D
96FD-   A9 97       LDA   #$97
96FF-   8D 8E 08    STA   $088E

; start the boot
9702-   4C 01 08    JMP   $0801

; (callback is here) relocate code to
; graphics page so it survives a reboot
9705-   A2 04       LDX   #$04
9707-   B9 00 BB    LDA   $BB00,Y
970A-   99 00 2B    STA   $2B00,Y
970D-   C8          INY
970E-   D0 F7       BNE   $9707
9710-   EE 09 97    INC   $9709
9713-   EE 0C 97    INC   $970C
9716-   CA          DEX
9717-   D0 EE       BNE   $9707

; turn off slot 6 drive motor
9719-   AD E8 C0    LDA   $C0E8

; reboot to my work disk
971C-   4C 00 C5    JMP   $C500

*BSAVE TRACE,A$9600,L$11F
*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT1 BB00-BEFF,A$2B00,L$400
```

Chapter 2
Just Because You're Paranoid Doesn't Mean They're Not Out To Get You

My work disk runs Diversi-DOS 64K, which relocates almost all the DOS code to the language card to free up main memory. (Only $BF00+ is used.) That means I can put this bootloader where it expects to be, without disconnecting DOS first.

```
]BLOAD BOOT1 BB00-BEFF,A$BB00
]CALL -151

*BB29L

; set screen to text mode
BB29-   20 39 FB    JSR   $FB39

; reset I/O vectors
BB2C-   A9 F0       LDA   #$F0
BB2E-   85 36       STA   $36
BB30-   A9 FD       LDA   #$FD
BB32-   85 37       STA   $37
BB34-   85 39       STA   $39
BB36-   A9 1B       LDA   #$1B
BB38-   85 38       STA   $38
BB3A-   8D 00 E8    STA   $E800

; initialize all the things
BB3D-   A0 00       LDY   #$00
BB3F-   84 FD       STY   $FD
BB41-   84 F1       STY   $F1
BB43-   84 F2       STY   $F2

; $3000 is the slot number (x16)
; (saved in boot0)
BB45-   AD 00 30    LDA   $3000
BB48-   8D B8 BE    STA   $BEB8
```

```
; set vectors (BRK, reset, NMI, IRQ)
BB4B-    A9 5B       LDA   #$5B
BB4D-    8D F0 03    STA   $03F0
BB50-    8D F2 03    STA   $03F2
BB53-    8D FC 03    STA   $03FC
BB56-    8D FE 03    STA   $03FE
BB59-    A9 BE       LDA   #$BE
BB5B-    8D F1 03    STA   $03F1
BB5E-    8D F3 03    STA   $03F3
BB61-    8D FD 03    STA   $03FD
BB64-    8D FF 03    STA   $03FF
BB67-    49 A5       EOR   #$A5
BB69-    8D F4 03    STA   $03F4
BB6C-    A9 4C       LDA   #$4C
BB6E-    8D FB 03    STA   $03FB
BB71-    A9 FB       LDA   #$FB
BB73-    8D FA FF    STA   $FFFA
BB76-    8D FC FF    STA   $FFFC
BB79-    8D FE FF    STA   $FFFE
BB7C-    A9 03       LDA   #$03
BB7E-    8D FB FF    STA   $FFFB
BB81-    8D FD FF    STA   $FFFD
BB84-    8D FF FF    STA   $FFFF
```

That's a lot of paranoia right there.
Like, all the paranoia.

```
; Even more paranoia: check if the byte
; we wrote to the language card RAM
; ($E800, set at $BB3A) is still there
; after we switch back to ROM. If it
; is, that means that something (like a
; modified F8 PROM) is interfering with
; the ROM/RAM softswitches and we're
; better off leaving "ROM" enabled
; because it's more likely to actually
; have the modifications we just made
; to all the low-level vectors at
; $FFFA..$FFFF. Out-faking the fakers.
BB87-    AD 80 C0     LDA   $C080
BB8A-    AD 00 E8     LDA   $E800
BB8D-    C9 1B        CMP   #$1B
BB8F-    F0 03        BEQ   $BB94
BB91-    8D 81 C0     STA   $C081

; clear text screen 1
BB94-    A9 A0        LDA   #$A0
BB96-    99 00 04     STA   $0400,Y
BB99-    99 00 05     STA   $0500,Y
BB9C-    99 00 06     STA   $0600,Y
BB9F-    99 00 07     STA   $0700,Y
BBA2-    C8           INY
BBA3-    D0 F1        BNE   $BB96

; show text screen 1 (blank)
BBA5-    AD 51 C0     LDA   $C051
BBA8-    AD 54 C0     LDA   $C054

; and now this
BBAB-    A9 02        LDA   #$02
BBAD-    A0 05        LDY   #$05
BBAF-    8D 03 BB     STA   $BB03
BBB2-    8C 02 BB     STY   $BB02
BBB5-    20 E7 BB     JSR   $BBE7
```

```
*BBE7L

BBE7-   48              PHA
BBE8-   4A              LSR
BBE9-   A8              TAY
BBEA-   A9 00           LDA   #$00
BBEC-   85 F7           STA   $F7
BBEE-   B9 5B BC        LDA   $BC5B,Y
BBF1-   85 F8           STA   $F8
BBF3-   68              PLA
```

Now ($F7) points to a page in memory. Which page depends on the value of the accumulator on entry, divided by 2, used as an index into the array at $BC5B.

```
BC5B- BB 08 14 20 2C
         ^^
     first page = $0800

BC60- 38 44 50 5C 68 74 80 8C
BC68- 98 A4 AF A9 42 A6 FB 20
               |-------------|
               real code, not
              part of the array
```

So it appears that ($F7) will initially point to $0800, since A=2 on entry. If A=4, it would point to $1400, and so on up to $AF00. In other words, the loader will be filling most of main memory $0C pages at a time, starting at $0800 and going up to the bootloader that lives at $BB00.

```
; (not shown) this moves the drive head
; to the specified phase (track x 2) --
; so track 1, since A=2
BBF4-    A6 FB        LDX   $FB
BBF6-    20 9A BC     JSR   $BC9A

; find a custom prologue "AD DA DD"
BBF9-    A9 0C        LDA   #$0C
BBFB-    85 FA        STA   $FA
BBFD-    BD 8E C0     LDA   $C08E,X
BC00-    BD 8C C0     LDA   $C08C,X
BC03-    10 FB        BPL   $BC00
BC05-    C9 AD        CMP   #$AD
BC07-    D0 F7        BNE   $BC00
BC09-    BD 8C C0     LDA   $C08C,X
BC0C-    10 FB        BPL   $BC09
BC0E-    C9 DA        CMP   #$DA
BC10-    D0 F3        BNE   $BC05
BC12-    BD 8C C0     LDA   $C08C,X
BC15-    10 FB        BPL   $BC12
BC17-    C9 DD        CMP   #$DD
BC19-    D0 EA        BNE   $BC05
```

```
; calculate a running checksum and read
; 4-4 encoded data into ($F7)
BC1B-    A0 00       LDY   #$00
BC1D-    84 F5       STY   $F5
BC1F-    BD 8C C0    LDA   $C08C,X
BC22-    10 FB       BPL   $BC1F
BC24-    38          SEC
BC25-    2A          ROL
BC26-    85 F6       STA   $F6
BC28-    4C 3C BC    JMP   $BC3C
BC2B-    BD 8C C0    LDA   $C08C,X
BC2E-    10 FB       BPL   $BC2B
BC30-    2A          ROL
BC31-    85 F6       STA   $F6
BC33-    C8          INY
BC34-    D0 06       BNE   $BC3C
BC36-    E6 F8       INC   $F8
BC38-    C6 FA       DEC   $FA
BC3A-    F0 10       BEQ   $BC4C
BC3C-    BD 8C C0    LDA   $C08C,X
BC3F-    10 FB       BPL   $BC3C
BC41-    25 F6       AND   $F6
BC43-    91 F7       STA   ($F7),Y
BC45-    45 F5       EOR   $F5
BC47-    85 F5       STA   $F5
BC49-    4C 2B BC    JMP   $BC2B

; verify checksum
BC4C-    BD 8C C0    LDA   $C08C,X
BC4F-    10 FB       BPL   $BC4C
BC51-    25 F6       AND   $F6
BC53-    45 F5       EOR   $F5
BC55-    D0 02       BNE   $BC59

; clear carry on success
BC57-    18          CLC
BC58-    60          RTS
```

```
; set carry on failure
BC59-   38            SEC
BC5A-   60            RTS
```

Continuing from $BBB8...

```
BBB8-   AD 03 BB      LDA   $BB03

; carry is clear on success
BBBB-   90 18         BCC   $BBD5

; retry read on failure ($BB02 is a
; global number of retries across the
; entire disk -- if it hits 0, the disk
; is considered bad and it jumps to The
; Badlands)
BBBD-   AC 02 BB      LDY   $BB02
BBC0-   88            DEY
BBC1-   F0 0F         BEQ   $BBD2
BBC3-   8C 02 BB      STY   $BB02
BBC6-   20 1F BB      JSR   $BB1F
BBC9-   AC 02 BB      LDY   $BB02
BBCC-   AD 03 BB      LDA   $BB03
BBCF-   4C AF BB      JMP   $BBAF

; too many retries (from $BBC1) means
; we jump to The Badlands
BBD2-   4C 56 BE      JMP   $BE56
```

```
*BE56L

BE56-   A9 B9         LDA   #$B9
BE58-   4C 5D BE      JMP   $BE5D
BE5B-   A9 C3         LDA   #$C3
BE5D-   85 F7         STA   $F7
BE5F-   A9 BE         LDA   #$BE
BE61-   85 F8         STA   $F8
```

```
; show text page
BE63-    AD 51 C0       LDA   $C051
BE66-    AD 54 C0       LDA   $C054

; clear text page
BE69-    A9 00          LDA   #$00
BE6B-    85 F3          STA   $F3
BE6D-    A9 A0          LDA   #$A0
BE6F-    A2 04          LDX   #$04
BE71-    86 F4          STX   $F4
BE73-    A0 0A          LDY   #$0A
BE75-    91 F3          STA   ($F3),Y
BE77-    C8             INY
BE78-    D0 FB          BNE   $BE75
BE7A-    E6 F4          INC   $F4
BE7C-    CA             DEX
BE7D-    D0 F6          BNE   $BE75

; display an error message in the upper
; left corner of the screen
BE7F-    A0 09          LDY   #$09
BE81-    B1 F7          LDA   ($F7),Y
BE83-    99 00 04       STA   $0400,Y
BE86-    88             DEY
BE87-    10 F8          BPL   $BE81
BE89-    C8             INY
BE8A-    84 00          STY   $00
```

```
; wipe main memory
BE8C-   A2 B7       LDX   #$B7
BE8E-   A9 BE       LDA   #$BE
BE90-   85 01       STA   $01
BE92-   98          TYA
BE93-   A0 55       LDY   #$55
BE95-   91 00       STA   ($00),Y
BE97-   88          DEY
BE98-   D0 FB       BNE   $BE95
BE9A-   C6 01       DEC   $01
BE9C-   CA          DEX
BE9D-   D0 F6       BNE   $BE95

; play a cute sound
BE9F-   A2 40       LDX   #$40
BEA1-   A9 14       LDA   #$14
BEA3-   20 A8 FC    JSR   $FCA8
BEA6-   AD 30 C0    LDA   $C030
BEA9-   CA          DEX
BEAA-   D0 F5       BNE   $BEA1
BEAC-   A9 00       LDA   #$00
BEAE-   20 A8 FC    JSR   $FCA8
BEB1-   E8          INX
BEB2-   E0 06       CPX   #$06
BEB4-   D0 F6       BNE   $BEAC

; and reboot
BEB6-   4C 00 C6    JMP   $C600
```

Continuing from $BBD5...

```
; execution continues here after a
; successful read (from $BBBB) --
; increment the phase and branch back
; to read a hard-coded number of tracks
BBD5-   69 02       ADC   #$02
BBD7-   C9 20       CMP   #$20
BBD9-   D0 D2       BNE   $BBAD
```

By the time we fall through here, we've run through the entire array at $BC5B and filled most of main memory ($0800..$BAFF) with game code.

```
; don't know what these are yet
BBDB-    20 6B BC     JSR   $BC6B
BBDE-    20 23 BD     JSR   $BD23

; turn off drive motor
BBE1-    BD 88 C0     LDA   $C088,X
BBE4-    4C CD BE     JMP   $BECD

*BECDL

; wipe bootloader from memory
BECD-    A0 00        LDY   #$00
BECF-    98           TYA
BED0-    99 00 BB     STA   $BB00,Y
BED3-    99 00 BC     STA   $BC00,Y
BED6-    99 00 BD     STA   $BD00,Y
BED9-    C8           INY
BEDA-    D0 F4        BNE   $BED0

; start the game
BEDC-    4C 00 08     JMP   $0800
```

That just leaves the mystery routines at $BC6B and $BD23.

Chapter 3
In Which We Detect The Matrix From Inside The Matrix

```
*BC6BL

; move drive head to track $21
BC6B-   A9 42       LDA   #$42
BC6D-   A6 FB       LDX   $FB
BC6F-   20 9A BC    JSR   $BC9A

; look for an $AA nibble, then count
; nibbles until another $AA
BC72-   BD 8E C0    LDA   $C08E,X
BC75-   BD 8C C0    LDA   $C08C,X
BC78-   10 FB       BPL   $BC75
BC7A-   C9 AA       CMP   #$AA
BC7C-   D0 F7       BNE   $BC75
BC7E-   BD 8C C0    LDA   $C08C,X
BC81-   10 FB       BPL   $BC7E
BC83-   BD 8C C0    LDA   $C08C,X
BC86-   10 FB       BPL   $BC83
BC88-   BD 8C C0    LDA   $C08C,X
BC8B-   10 FB       BPL   $BC88
BC8D-   C9 AA       CMP   #$AA
BC8F-   F0 08       BEQ   $BC99
BC91-   E6 F1       INC   $F1
BC93-   D0 F3       BNE   $BC88
BC95-   E6 F2       INC   $F2
BC97-   D0 EF       BNE   $BC88
BC99-   60          RTS

*BD23L

; move drive head to track $22
BD23-   A9 44       LDA   #$44
BD25-   A6 FB       LDX   $FB
BD27-   20 9A BC    JSR   $BC9A
```

```
; initialize counters
BD2A-    A9 04          LDA   #$04
BD2C-    85 12          STA   $12
BD2E-    A9 00          LDA   #$00
BD30-    85 11          STA   $11
BD32-    A9 08          LDA   #$08
BD34-    85 FE          STA   $FE
BD36-    A0 00          LDY   #$00
BD38-    84 10          STY   $10

; look for a long nibble sequence
; "AA D5 D5 FF D6 FF FD FD DD EA B5 F7"
BD3A-    BD 8C C0       LDA   $C08C,X
BD3D-    10 FB          BPL   $BD3A
BD3F-    C9 AA          CMP   #$AA
BD41-    D0 F7          BNE   $BD3A
BD43-    BD 8C C0       LDA   $C08C,X
BD46-    10 FB          BPL   $BD43
BD48-    C9 D5          CMP   #$D5
BD4A-    D0 F3          BNE   $BD3F
BD4C-    BD 8C C0       LDA   $C08C,X
BD4F-    10 FB          BPL   $BD4C
BD51-    C9 D5          CMP   #$D5
BD53-    D0 EA          BNE   $BD3F
BD55-    BD 8C C0       LDA   $C08C,X
BD58-    10 FB          BPL   $BD55
BD5A-    C9 FF          CMP   #$FF
BD5C-    D0 E1          BNE   $BD3F
BD5E-    BD 8C C0       LDA   $C08C,X
BD61-    10 FB          BPL   $BD5E
BD63-    C9 D6          CMP   #$D6
BD65-    D0 D8          BNE   $BD3F
BD67-    BD 8C C0       LDA   $C08C,X
BD6A-    10 FB          BPL   $BD67
BD6C-    C9 FF          CMP   #$FF
BD6E-    D0 CF          BNE   $BD3F
BD70-    BD 8C C0       LDA   $C08C,X
BD73-    10 FB          BPL   $BD70
[...]
```

```
BD75-    C9 FD       CMP   #$FD
BD77-    D0 C6       BNE   $BD3F
BD79-    BD 8C C0    LDA   $C08C,X
BD7C-    10 FB       BPL   $BD79
BD7E-    C9 FD       CMP   #$FD
BD80-    D0 BD       BNE   $BD3F
BD82-    BD 8C C0    LDA   $C08C,X
BD85-    10 FB       BPL   $BD82
BD87-    C9 DD       CMP   #$DD
BD89-    D0 B4       BNE   $BD3F
BD8B-    BD 8C C0    LDA   $C08C,X
BD8E-    10 FB       BPL   $BD8B
BD90-    C9 EA       CMP   #$EA
BD92-    D0 AB       BNE   $BD3F
BD94-    BD 8C C0    LDA   $C08C,X
BD97-    10 FB       BPL   $BD94
BD99-    C9 B5       CMP   #$B5
BD9B-    D0 A2       BNE   $BD3F
BD9D-    BD 8C C0    LDA   $C08C,X
BDA0-    10 FB       BPL   $BD9D
BDA2-    C9 F7       CMP   #$F7
BDA4-    D0 99       BNE   $BD3F
```

```
; decode two bytes of 4-4 encoded data,
; store them in zero page $F6 and $FA
BDA6-   BD 8C C0     LDA   $C08C,X
BDA9-   10 FB        BPL   $BDA6
BDAB-   38           SEC
BDAC-   2A           ROL
BDAD-   85 F6        STA   $F6
BDAF-   BD 8C C0     LDA   $C08C,X
BDB2-   10 FB        BPL   $BDAF
BDB4-   25 F6        AND   $F6
BDB6-   85 F6        STA   $F6
BDB8-   BD 8C C0     LDA   $C08C,X
BDBB-   10 FB        BPL   $BDB8
BDBD-   38           SEC
BDBE-   2A           ROL
BDBF-   85 FA        STA   $FA
BDC1-   BD 8C C0     LDA   $C08C,X
BDC4-   10 FB        BPL   $BDC1
BDC6-   25 FA        AND   $FA
BDC8-   85 FA        STA   $FA

; compute a rolling checksum on a long
; sequence of nibbles
BDCA-   BD 8C C0     LDA   $C08C,X
BDCD-   10 FB        BPL   $BDCA
BDCF-   BD 8C C0     LDA   $C08C,X
BDD2-   10 FB        BPL   $BDCF
BDD4-   45 10        EOR   $10
BDD6-   85 10        STA   $10
BDD8-   C8           INY
BDD9-   D0 F4        BNE   $BDCF
BDDB-   C6 FE        DEC   $FE
BDDD-   D0 F0        BNE   $BDCF

; calculate a second rolling checksum
; from the final value of the first
; rolling checksum
BDDF-   A5 10        LDA   $10
BDE1-   45 11        EOR   $11
BDE3-   85 11        STA   $11
```

```
; loop back and do it again, a total of
; 4 times (zero page $12 set at $BD2C)
BDE5-   C6 12       DEC   $12
BDE7-   F0 03       BEQ   $BDEC
BDE9-   4C 32 BD    JMP   $BD32

; check secondary checksum
BDEC-   A5 11       LDA   $11

; needs to be non-zero
BDEE-   D0 03       BNE   $BDF3

; ...or we jump to The Badlands
BDF0-   4C 56 BE    JMP   $BE56

; look for an $F6 nibble
BDF3-   A9 00       LDA   #$00
BDF5-   85 00       STA   $00
BDF7-   A9 18       LDA   #$18
BDF9-   85 01       STA   $01
BDFB-   BD 8C C0    LDA   $C08C,X
BDFE-   10 FB       BPL   $BDFB
BE00-   C9 F6       CMP   #$F6
BE02-   F0 0B       BEQ   $BE0F
BE04-   C6 00       DEC   $00
BE06-   D0 F3       BNE   $BDFB
BE08-   C6 01       DEC   $01
BE0A-   D0 EF       BNE   $BDFB

; if we don't find the $F6 nibble, it's
; off to The Badlands for you!
BE0C-   4C 56 BE    JMP   $BE56
```

```
; execution continues here (from $BE00)
; once we find the first $F6 nibble --
; now
BE0F-   A0 80       LDY   #$80
BE11-   BD 8C C0    LDA   $C08C,X
BE14-   10 FB       BPL   $BE11
BE16-   C9 F6       CMP   #$F6
BE18-   D0 E1       BNE   $BDFB
BE1A-   88          DEY
BE1B-   D0 F4       BNE   $BE11

; increment $F1/$F2 counter (set from
; the nibble counting on track $21)
BE1D-   20 39 BE    JSR   $BE39

; check if that counter equals the
; values we read from track $22 (into
; zero page $F6 and $FA, at $BDA6)
BE20-   20 4B BE    JSR   $BE4B

; if they match, we're done
BE23-   F0 10       BEQ   $BE35

; decrement the $F1/$F2 counter
BE25-   20 40 BE    JSR   $BE40

; check again if $F1/$F2 == $F6/$FA
BE28-   20 4B BE    JSR   $BE4B

; if they match, we're done
BE2B-   F0 08       BEQ   $BE35

; decrement $F1/$F2 one more time and
; re-check
BE2D-   20 40 BE    JSR   $BE40
BE30-   20 4B BE    JSR   $BE4B
```

```
; still no luck, jump to The Badlands
BE33-   D0 01        BNE   $BE36
BE35-   60           RTS
BE36-   4C 56 BE     JMP   $BE56

BE39-   E6 F1        INC   $F1
BE3B-   D0 18        BNE   $BE55
BE3D-   E6 F2        INC   $F2
BE3F-   60           RTS

BE40-   C6 F1        DEC   $F1
BE42-   A5 F1        LDA   $F1
BE44-   C9 FF        CMP   #$FF
BE46-   D0 0D        BNE   $BE55
BE48-   C6 F2        DEC   $F2
BE4A-   60           RTS

BE4B-   A5 F6        LDA   $F6
BE4D-   C5 F1        CMP   $F1
BE4F-   D0 04        BNE   $BE55
BE51-   A5 FA        LDA   $FA
BE53-   C5 F2        CMP   $F2
BE55-   60           RTS
```

There are several things going on here. The easiest one to grok is the nibble counting and matching of those counters between tracks $21 and $22. But there is a much more interesting part to this protection, which boils down to

1. find a long nibble prologue (that only appears once on the track)
2. checksum the following nibbles
3. do steps 1 and 2 repeatedly and make sure the checksum changes

This is the key point: the data being read from track $22 is non-repeatable. It's different every time it's read. How is that possible?

The prologue ("AA D5 D5 FF D6 FF FD FD DD EA B5 F7") looks important, but it's not. What's important is what comes after it, what's being checksummed over and over: a long sequence of zero bits. Because that is what is actually on the original disk: nothing.

When we say a "zero bit," we really mean "the lack of a magnetic state change." If the Disk II doesn't see a state change in a certain period of time, it calls that a "0". If it does see a change, it calls that a "1". But the drive can only tolerate a lack of state changes for so long -- about as long as it takes for two bits to go by.

Fun fact(*): this is why you need to use nibbles as an intermediate on-disk format in the first place. No valid nibble contains more than two zero bits consecutively, when written from most-significant to least-significant bit.

(*) not guaranteed, actual fun may vary

So what happens when a drive doesn't see a state change after the equivalent of two consecutive zero bits? The drive thinks the disk is weak, and it starts increasing the amplification to try to compensate, looking for a valid signal. But there is no signal. There is no data. There is just a yawning abyss of nothingness. Eventually, the drive gets desperate and amplifies so much that it starts returning random bits based on ambient noise from the disk motor and the magnetism of the Earth.

Seriously.

Returning random bits doesn't sound very useful for a storage medium, but it's exactly what the developer wanted, and that's exactly what this code is checking for. It's finding and reading and checksumming the same sequence of bits from the disk, over and over, and checking that they change every time.

Bit copiers will never duplicate the long sequence of zero bits, because that's not what they read. Whatever randomness they get when they read the original disk will essentially get "frozen" onto the copy. The checksum of those frozen bits will always be the same, no matter how many times you read them. So the BNE at $BDEE will never branch, and it will fall through to $BDF0 and jump to The Badlands.

God, I hate physical objects.

Chapter 4
In Which We Enter The Home Stretch

I can let the protection check pass (and it will, since I'm working from an original disk), then patch the JMP at $BEDC to capture the game in memory before it starts.

```
*9600<C600.C6FFM

; set up callback to my code after RWTS
; is loaded into $BB00
96F8-   A9 05       LDA   #$05
96FA-   8D 8D 08    STA   $088D
96FD-   A9 97       LDA   #$97
96FF-   8D 8E 08    STA   $088E
9702-   4C 01 08    JMP   $0801

; unconditionally break to monitor
; instead of starting the game at $BEDC
9705-   A9 59       LDA   #$59
9707-   8D DD BE    STA   $BEDD
970A-   A9 FF       LDA   #$FF
970C-   8D DE BE    STA   $BEDE

; continue the boot
970F-   4C 29 BB    JMP   $BB29

*BSAVE TRACE2,A$9600,L$112
*9600G
...reboots slot 6...
...read read read...
<beep>
```

Success! The entire game is in memory.

```
*800G
...works...
```

After a few judicious memory moves and reboots and BSAVEs and BRUN TRACE2, I have the entire game in several files on my work disk.

```
]PR#5
]CATALOG

C1983 DSR^C#254
264 FREE

 A 019 HELLO
 B 005 AUTOTRACE
 A 002 WAVY NAVY
 A 002 C 1982 SIRIUS
 B 002 MAKE
 B 003 BOOT0
 B 003 TRACE
 B 006 BOOT1 BB00-BEFF
 B 003 TRACE2
 B 026 OBJ.0800-1FFF
 B 066 OBJ.2000-5FFF
 B 066 OBJ.6000-9FFF
 B 029 OBJ.A000-BAFF

]BLOAD OBJ.2000-5FFF,A$2000
]BLOAD OBJ.6000-9FFF,A$6000
]BLOAD OBJ.A000-BAFF,A$A000
]BRUN OBJ.0800-1FFF
...works...
```

Chapter 5
Ha Ha, Just Kidding, We're Nowhere Near Done Yet

To reproduce the original disk's boot experience as faithfully as possible, I decided against releasing this as a file crack. It's 2016; nobody is trying to squeeze multiple games on a single floppy disk anymore. Let's write a fastloader.

First, we need to write the game to disk. We'll worry about reading it back in just a minute.

```
; page count (decremented)
0300-   A9 B3       LDA   #$B3
0302-   85 FF       STA   $FF

; logical sector (incremented)
0304-   A9 0D       LDA   #$0D
0306-   85 FE       STA   $FE

; call RWTS to write sector
0308-   A9 03       LDA   #$03
030A-   A0 88       LDY   #$88
030C-   20 D9 03    JSR   $03D9

; increment logical sector, wrap around
; from $0F to $00 and increment track
030F-   E6 FE       INC   $FE
0311-   A4 FE       LDY   $FE
0313-   C0 10       CPY   #$10
0315-   D0 07       BNE   $031E
0317-   A0 00       LDY   #$00
0319-   84 FE       STY   $FE
031B-   EE 8C 03    INC   $038C

; convert logical to physical sector
031E-   B9 40 03    LDA   $0340,Y
0321-   8D 8D 03    STA   $038D
```

```
; increment page to write
0324-   EE 91 03    INC   $0391

; loop until done with all pages
0327-   C6 FF       DEC   $FF
0329-   D0 DD       BNE   $0308
032B-   60          RTS

*340.34F

; logical to physical sector mapping
0340- 00 07 0E 06 0D 05 0C 04
0348- 0B 03 0A 02 09 01 08 0F

*388.397

; RWTS parameter table, pre-initialized
; with slot 6, drive 1, track $00,
; sector $01, address $0800, and RWTS
; write command ($02)
0388- 01 60 01 00 00 01 FB F7
0390- 00 08 00 00 02 00 00 60

*BSAVE MAKE,A$300,L$98

*300G          ; write game to disk
```

Boom. I have the entire game on tracks $00-$0B of a standard 16-sector disk. (The first 3 sectors are on track $00, then tracks $01-$0B are entirely full.)

Now, about that fastloader...

Once upon a time, I wrote a little thing called 4boot. It was fast and small and I was more than a little bit proud of it. The boot1 code was a mere 742 bytes and fit in $BD00..$BFFF.

Then qkumba did that thing he does, and now it fits in zero page.

With his blessing, I present: 0boot.

Chapter 6
0boot

0boot lives on track $00, just like me. Sector $00 (boot0) reuses the disk controller ROM routine to read sector $0E (boot1). Boot0 creates a few data tables, copys boot1 to zero page, modifies it to accomodate booting from any slot, and jumps to it.

Boot0 is loaded at $0800 by the disk controller ROM routine.

```
; tell the ROM to load only this sector
; (we'll do the rest manually)
0800-   [01]

; The accumulator is $01 after loading
; sector $00, or $03 after loading
; sector $0E. We don't need to preserve
; the value, so we just shift the bits
; to determine whether this is the
; first or second time we've been here.
0801-    4A             LSR

; second run -- we've loaded boot1, so
; skip to boot1 initialization routine
0802-    D0 0E          BNE   $0812

; first run -- increment the physical
; sector to read (this will be the next
; sector under the drive head, so we'll
; waste as little time as possible
; waiting for the disk to spin)
0804-    E6 3D          INC   $3D
```

```
; X holds the boot slot (x16) --
; munge it into $Cx format (e.g. $C6
; for slot 6, but we need to accomodate
; booting from any slot)
0806-   8A          TXA
0807-   4A          LSR
0808-   4A          LSR
0809-   4A          LSR
080A-   4A          LSR
080B-   09 C0       ORA   #$C0

; push address (-1) of the sector read
; routine in the disk controller ROM
080D-   48          PHA
080E-   A9 5B       LDA   #$5B
0810-   48          PHA

; "return" via disk controller ROM,
; which reads boot1 into $0900 and
; exits via $0801
0811-   60          RTS

; Execution continues here (from $0802)
; after boot1 code has been loaded into
; $0900. This works around a bug in the
; CFFA 3000 firmware that doesn't
; guarantee that the Y register is
; always $00 at $0801, which is exactly
; the sort of bug that qkumba enjoys(*)
; uncovering.
0812-   A8          TAY


; munge the boot slot, e.g. $60 -> $EC
; (to be used later)
0813-   8A          TXA
0814-   09 8C       ORA   #$8C
```

(*) not guaranteed, actual enjoyment
    may vary

```
; Copy the boot1 code from $0901..$09FF
; to zero page. ($0900 holds the 0boot
; version number. This is version 1.
; $0000 is initialized later in boot1.)
0816-    BE 00 09       LDX   $0900,Y
0819-    96 00          STX   $00,Y
081B-    C8             INY
081C-    D0 F8          BNE   $0816

; There are a number of places in boot1
; that need to hit a slot-specific soft
; switch (read a nibble from disk, turn
; off the drive, &c). Rather than the
; usual form of "LDA $C08C,X", we will
; use "LDA $C0EC" and modify the $EC
; byte in advance, based on the boot
; slot. $00F5 is an array of all the
; places in the boot1 code that need
; this adjustment.
081E-    C8             INY
081F-    B6 F5          LDX   $F5,Y
0821-    95 00          STA   $00,X
0823-    D0 F9          BNE   $081E

; munge $EC -> $E0 (used later to
; advance the drive head to the next
; track)
0825-    29 F0          AND   #$F0
0827-    85 C8          STA   $C8

; munge $E0 -> $E8 (used later to
; turn off the drive motor)
0829-    09 08          ORA   #$08
082B-    85 D6          STA   $D6
```

```
; push several addresses to the stack
; (more on this later)
082D-   A2 0C       LDX   #$0C
082F-   B5 E9       LDA   $E9,X
0831-   48          PHA
0832-   CA          DEX
0833-   D0 FA       BNE   $082F

; number of tracks to load (x2) (game-
; specific -- this game uses $B tracks,
; not including the sprinking of
; sectors on track $00)
0835-   A0 16       LDY   #$16

; loop starts here
083C-   8A          TXA

; every other time through this loop,
; we will end up taking this branch
083D-   90 03       BCC   $0842

; X is 0 going into this loop, and it
; never changes, so A is always 0 too.
; So this will push $0000 to the stack
; (to "return" to $0001, which reads a
; track into memory)
083F-   48          PHA
0840-   48          PHA
```

```
; There's a "SEC" hidden here (because
; it's opcode $38), but it's only
; executed if we take the branch at
; $0840, which lands at $0845, which is
; in the middle of this instruction.
; Otherwise we execute the compare,
; which clears the carry bit. So the
; carry flip-flops between set and
; clear, so the BCC at $0840 is only
; taken every other time.
0841-   C9 38        CMP   #$38
0843-   48           PHA

; Push $00B3 to the stack, to "return"
; to $00B4. This routine advances the
; drive head to the next half track.
0844-   A9 B3        LDA   #$B3
0846-   48           PHA

; loop until done
0847-   88           DEY
0848-   D0 F2        BNE   $083C
```

Because of the carry flip-flop, we will push $00B3 to the stack every time through the loop, but we will only push $0000 every other time. The loop runs for twice the number of tracks we want to read, so the stack ends up looking like this:

```
 --top--
  $00B3 (move drive 1/2 track)
  $00B3 (move drive another 1/2 track)
  $0000 (read track into memory)
  $00B3 \
  $00B3  } second group
  $0000 /
  $00B3 \
  $00B3  } third group
  $0000 /
    .
    . [repeated for each track]
    .
  $00B3 \
  $00B3  } final group
  $0000 /
  $FE88 (IN#0, pushed at $0831)
  $FE92 (PR#0, pushed at $0831)
  $FE83 (NORMAL, pushed at $0831)
  $FB2E (TEXT, pushed at $0831)
  $00D4 (turn off drive motor)
  $07FF (game entry point)
--bottom--
```

Boot1 reads the game into memory from tracks $00-$0B, but it isn't a loop. It's one routine that reads a track and another routine that advances the drive head. We're essentially unrolling the read loop on the stack, in advance, so that each routine gets called as many times as we need, when we need it. Like dancers in a chorus line, each routine executes then cedes the spotlight. Each seems unaware of the others, but in reality they've all been meticulously choreographed.

Chapter 7
6 + 2

Before I can explain the next chunk of code, I need to pause and explain a little bit of theory. As you probably know if you're the sort of person who reads this sort of thing, Apple II floppy disks do not contain the actual data that ends up being loaded into memory. Due to hardware limitations of the original Disk II drive, data on disk must be stored in an intermediate format called "nibbles." Bytes in memory are encoded into nibbles before writing to disk, and nibbles that you read from the disk must be decoded back into bytes. The round trip is lossless but requires some bit wrangling.

Decoding nibbles-on-disk into bytes-in-memory is a multi-step process. In "6-and-2 encoding" (used by DOS 3.3, ProDOS, and all ".dsk" image files), there are 64 possible values that you may find in the data field (in the range $96..$FF, but not all of those, because some of them have bit patterns that trip up the drive firmware). We'll call these "raw nibbles."

Step 1: read $156 raw nibbles from the data field. These values will range from $96 to $FF, but as mentioned earlier, not all values in that range will appear on disk.

Now we have $156 raw nibbles.

Step 2: decode each of the raw nibbles into a 6-bit byte between 0 and 63 (%00000000 and %00111111 in binary). $96 is the lowest valid raw nibble, so it gets decoded to 0. $97 is the next valid raw nibble, so it's decoded to 1. $98 and $99 are invalid, so we skip them, and $9A gets decoded to 2. And so on, up to $FF (the highest valid raw nibble), which gets decoded to 63.

Now we have $156 6-bit bytes.

Step 3: split up each of the first $56 6-bit bytes into pairs of bits. In other words, each 6-bit byte becomes three 2-bit bytes. These 2-bit bytes are merged with the next $100 6-bit bytes to create $100 8-bit bytes. Hence the name, "6-and-2" encoding.

The exact process of how the bits are split and merged is... complicated. The first $56 6-bit bytes get split up into 2-bit bytes, but those two bits get swapped (so %01 becomes %10 and vice-versa). The other $100 6-bit bytes each get multiplied by 4 (a.k.a. bit-shifted two places left). This leaves a hole in the lower two bits, which is filled by one of the 2-bit bytes from the first group.

A diagram might help. "a" through "x" each represent one bit.

```
             -------------


1 decoded      3 decoded
nibble in  +   nibbles in   =  3 bytes
first $56      other $100


00abcdef       00ghijkl
               00mnopqr
   |           00stuvwx
   |
 split            |
   &           shifted
swapped        left x2
   |              |
   V              V

000000fe   +   ghijkl00   =   ghijklfe
000000dc   +   mnopqr00   =   mnoprqdc
000000ba   +   stuvwx00   =   stuvwxba

             -------------
```

Tada! Four 6-bit bytes

```
  00abcdef
  00ghijkl
  00mnopqr
  00stuvwx
```

become three 8-bit bytes

```
  ghijklfe
  mnoprqdc
  stuvwxba
```

When DOS 3.3 reads a sector, it reads the first $56 raw nibbles, decoded them into 6-bit bytes, and stashes them in a temporary buffer (at $BC00). Then it reads the other $100 raw nibbles, decodes them into 6-bit bytes, and puts them in another temporary buffer (at $BB00). Only then does DOS 3.3 start combining the bits from each group to create the full 8-bit bytes that will end up in the target page in memory. This is why DOS 3.3 "misses" sectors when it's reading, because it's busy twiddling bits while the disk is still spinning.

Chapter 8
Back to 0boot

0boot also uses "6-and-2" encoding. The first $56 nibbles in the data field are still split into pairs of bits that need to be merged with nibbles that won't come until later. But instead of waiting for all $156 raw nibbles to be read from disk, it "interleaves" the nibble reads with the bit twiddling required to merge the first $56 6-bit bytes and the $100 that follow. By the time 0boot gets to the data field checksum, it has already stored all $100 8-bit bytes in their final resting place in memory. This means that 0boot can read all 16 sectors on a track in one revolution of the disk. That's crazy fast.

To make it possible to do all the bit twiddling we need to do and not miss nibbles as the disk spins(*), we do some of the work earlier. We multiply each of the 64 possible decoded values by 4 and store those values. (Since this is accomplished by bit shifting and we're doing it before we start reading the disk, this is called the "pre-shift" table.) We also store all possible 2-bit values in a repeating pattern that will make it easy to look them up later. Then, as we're reading from disk (and timing is tight), we can simulate all the bit math we need to do with a series of table lookups. There is just enough time to convert each raw nibble into its final 8-bit byte before reading the next nibble.

(*) The disk spins independently of the CPU, and we only have a limited time to read a nibble and do what we're going to do with it before WHOOPS HERE COMES ANOTHER ONE. So time is of the essence. Also, "As The Disk Spins" would make a great name for a retrocomputing-themed soap opera.

The first table, at $0200..$02FF, is three columns wide and 64 rows deep. Astute readers will notice that 3 x 64 is not 256. Only three of the columns are used; the fourth (unused) column exists because multiplying by 3 is hard but multiplying by 4 is easy (in base 2 anyway). The three columns correspond to the three pairs of 2-bit values in those first $56 6-bit bytes. Since the values are only 2 bits wide, each column holds one of four different values (%00, %01, %10, or %11).

The second table, at $0300..$0369, is the "pre-shift" table. This contains all the possible 6-bit bytes, in order, each multiplied by 4 (a.k.a. shifted to the left two places, so the 6 bits that started in columns 0-5 are now in columns 2-7, and columns 0 and 1 are zeroes). Like this:

```
       00ghijkl   -->   ghijkl00
```

Astute readers will notice that there are only 64 possible 6-bit bytes, but this second table is larger than 64 bytes. To make lookups easier, the table has empty slots for each of the invalid raw nibbles. In other words, we don't do any math to decode raw nibbles into 6-bit bytes; we just look them up in this table (offset by $96, since that's the lowest valid raw nibble) and get the required bit shifting for free.

addr	raw	decoded 6-bit	pre-shift
$300	$96	0 = %00000000	%00000000
$301	$97	1 = %00000001	%00000100
$302	$98	[invalid raw nibble]	
$303	$99	[invalid raw nibble]	
$304	$9A	2 = %00000010	%00001000
$305	$9B	3 = %00000011	%00001100
$306	$9C	[invalid raw nibble]	
$307	$9D	4 = %00000100	%00010000
.			
.			
.			
$368	$FE	62 = %00111110	%11111000
$369	$FF	63 = %00111111	%11111100

Each value in this "pre-shift" table also serves as an index into the first table (with all the 2-bit bytes). This wasn't an accident; I mean, that sort of magic doesn't just happen. But the table of 2-bit bytes is arranged in such a way that we take one of the raw nibbles that needs to be decoded and split apart (from the first $56 raw nibbles in the data field), use that raw nibble as an index into the pre-shift table, then use that pre-shifted value as an index into the first table to get the 2-bit value we need. That's a neat trick.

```
; this loop creates the pre-shift table
; at $300
084A-    A2 40         LDX   #$40
084C-    A4 55         LDY   $55
084E-    98            TYA
084F-    0A            ASL
0850-    24 55         BIT   $55
0852-    F0 12         BEQ   $0866
0854-    05 55         ORA   $55
0856-    49 FF         EOR   #$FF
0858-    29 7E         AND   #$7E
085A-    B0 0A         BCS   $0866
085C-    4A            LSR
085D-    D0 FB         BNE   $085A
085F-    CA            DEX
0860-    8A            TXA
0861-    0A            ASL
0862-    0A            ASL
0863-    99 EA 02      STA   $02EA,Y
0866-    C6 55         DEC   $55
0868-    D0 E2         BNE   $084C
```

And this is the result (".." means the address is uninitialized and unused):

```
0300- 00 04 .. .. 08 0C .. 10
0308- 14 18 .. .. .. .. .. ..
0310- 1C 20 .. .. .. 24 28 2C
0318- 30 34 .. .. 38 3C 40 44
0320- 48 4C .. 50 54 58 5C 60
0328- 64 68 .. .. .. .. .. ..
0330- .. .. .. .. .. 6C .. 70
0338- 74 78 .. .. .. 7C .. ..
0340- 80 84 .. 88 8C 90 94 98
0348- 9C A0 .. .. .. .. .. A4
0350- A8 AC .. B0 B4 B8 BC C0
0358- C4 C8 .. .. CC D0 D4 D8
0360- DC E0 .. E4 E8 EC F0 F4
0368- F8 FC
```

```
; this loop creates the table of 2-bit
; values at $200, magically arranged to
; enable easy lookups later
086A-   46 B7          LSR   $B7
086C-   46 B7          LSR   $B7
086E-   B5 FC          LDA   $FC,X
0870-   99 FF 01       STA   $01FF,Y
0873-   E6 AC          INC   $AC
0875-   A5 AC          LDA   $AC
0877-   25 B7          AND   $B7
0879-   D0 05          BNE   $0880
087B-   E8             INX
087C-   8A             TXA
087D-   29 03          AND   #$03
087F-   AA             TAX
0880-   C8             INY
0881-   C8             INY
0882-   C8             INY
0883-   C8             INY
0884-   C0 04          CPY   #$04
0886-   B0 E6          BCS   $086E
0888-   C8             INY
0889-   C0 04          CPY   #$04
088B-   90 DD          BCC   $086A
```

And this is the result:

```
0200- 00 00 00 .. 00 00 02 ..
0208- 00 00 01 .. 00 00 03 ..
0210- 00 02 00 .. 00 02 02 ..
0218- 00 02 01 .. 00 02 03 ..
0220- 00 01 00 .. 00 01 02 ..
0228- 00 01 01 .. 00 01 03 ..
0230- 00 03 00 .. 00 03 02 ..
0238- 00 03 01 .. 00 03 03 ..
0240- 02 00 00 .. 02 00 02 ..
0248- 02 00 01 .. 02 00 03 ..
0250- 02 02 00 .. 02 02 02 ..
0258- 02 02 01 .. 02 02 03 ..
0260- 02 01 00 .. 02 01 02 ..
0268- 02 01 01 .. 02 01 03 ..
0270- 02 03 00 .. 02 03 02 ..
0278- 02 03 01 .. 02 03 03 ..
0280- 01 00 00 .. 01 00 02 ..
0288- 01 00 01 .. 01 00 03 ..
0290- 01 02 00 .. 01 02 02 ..
0298- 01 02 01 .. 01 02 03 ..
02A0- 01 01 00 .. 01 01 02 ..
02A8- 01 01 01 .. 01 01 03 ..
02B0- 01 03 00 .. 01 03 02 ..
02B8- 01 03 01 .. 01 03 03 ..
02C0- 03 00 00 .. 03 00 02 ..
02C8- 03 00 01 .. 03 00 03 ..
02D0- 03 02 00 .. 03 02 02 ..
02D8- 03 02 01 .. 03 02 03 ..
02E0- 03 01 00 .. 03 01 02 ..
02E8- 03 01 01 .. 03 01 03 ..
02F0- 03 03 00 .. 03 03 02 ..
02F8- 03 03 01 .. 03 03 03 ..
```

And now for something completely different. The original disk clears the text screen early and leaves it that way until the entire game is loaded. I can't call HOME ($FC58) at this point because it would destroy part of zero page which has actual code on it. So we get to do it the hard way.

```
088D-    A0 00        LDY   #$00
088F-    A9 A0        LDA   #$A0
0891-    99 00 04     STA   $0400,Y
0894-    99 00 05     STA   $0500,Y
0897-    99 00 06     STA   $0600,Y
089A-    99 00 07     STA   $0700,Y
089D-    C8           INY
089E-    D0 F1        BNE   $0891
```

Back to 0boot. Remember those 3 sectors on track $00? We're going to load those first, since, you know, we're already on track $00. But we need to clear the array that is used to track which sectors to load, so that we don't load any more than we need to. (This will, perhaps, make more sense once you see it in action in boot1.)

```
08A0-    A2 F0        LDX   #$F0
08A2-    A9 00        LDA   #$00
08A4-    95 00        STA   $00,X
08A6-    E8           INX
08A7-    E0 FD        CPX   #$FD
08A9-    D0 F9        BNE   $08A4

; now jump to the entry point to read
; the 3 sectors on track $00
08AB-    4C 03 00     JMP   $0003
```

[Note to future self: $088D..$08FD is available for game-specific init code, but it can't rely on or disturb zero page in any way. That rules out a lot of built-in ROM routines; be careful. If the game needs no initialization, you can zap this entire range and put an "RTS" at $088D.]

At this point, boot0 is done. We jumped (not JSR'd) to read the sectors on track $00. So what happens next? Well, everything else is already lined up on the stack. All that's left to do is "return" and let the stack guide us through the rest of the boot.

Chapter 9
0boot boot1

The rest of the boot runs from zero page. It's hard to show you exactly what boot1 will look like, because it relies heavily on self-modifying code.

In a standard DOS 3.3 RWTS, the softswitch to read the data latch is "LDA $C08C,X", where X is the boot slot times 16 (to allow disks to boot from any slot). 0boot also supports booting from any slot, but instead of using an index, each fetch instruction is pre-set based on the boot slot. Not only does this free up the X register, it lets us juggle all the registers and put the raw nibble value in whichever one is convenient at the time. (We take full advantage of this freedom.) I've marked each pre-set softswitch with "o_O" to remind you that self-modifying code is awesome.

There are several other instances of addresses and constants that get modified while boot1 is running. I've marked these with "/!\" to remind you that self-modifying code is dangerous and you should not try this at home.

The first thing popped off the stack is the drive arm move routine at $00B4. It moves the drive exactly one phase (half a track).

```
00B4-   E6 B7        INC   $B7
```

```
; This value was set at $00B4 (above).
; It's incremented monotonically, but
; it's ANDed with $03 later, so its
; exact value isn't relevant.
00B6-   A0 00       LDY   #$00      /!\

; short wait for PHASEON
00B8-   A9 04       LDA   #$04
00BA-   20 C0 00    JSR   $00C0

; fall through
00BD-   88          DEY

; longer wait for PHASEOFF
00BE-   69 41       ADC   #$41
00C0-   85 CB       STA   $CB

; calculate the proper stepper motor to
; access
00C2-   98          TYA
00C3-   29 03       AND   #$03
00C5-   2A          ROL
00C6-   AA          TAX

; This address was set at $0827,
; based on the boot slot.
00C7-   BD E0 C0    LDA   $C0E0,X   /!\

; This value was set at $00C0 so that
; PHASEON and PHASEOFF have optimal
; wait times.
00CA-   A9 D1       LDA   #$D1      /!\

; wait exactly the right amount of time
; after accessing the proper stepper
; motor
00CC-   4C A8 FC    JMP   $FCA8
```

Since the drive arm routine only moves one phase, it was pushed to the stack twice before each track read. Our game is stored on whole tracks; this half-track trickery is only to save a few bytes of code in boot1.

The track read routine starts at $0001, because that let us save 1 byte in the boot0 code when we were pushing addresses to the stack. (We could just push $00 twice.)

```
; sectors-left-to-read-on-this-track
; counter (incremented to $00)
0001-   A2 F0       LDX   #$F0
0003-   86 00       STX   $00
```

We initialize an array at $00F0 that tracks which sectors we've read from the current track. Astute readers will notice that this part of zero page had real data in it -- some addresses that were pushed to the stack, and some other values that were used to create the 2-bit table at $0200. All true, but all those operations are now complete, and the space from $00F0..$00FF is now available for unrelated uses.

The array is in physical sector order, thus the RWTS assumes data is stored in physical sector order on each track. (This is why my MAKE program had to map to physical sector order when writing. This saves 18 bytes: 16 for the table and 2 for the lookup command!) Values are the actual pages in memory where that sector should go, and they get zeroed once the sector is read (so we don't waste time decoding the same sector twice).

```
; starting address (game-specific;
; this one starts loading at $0A00)
0005-   A9 0A         LDA   #$0A          /!\
0007-   95 00         STA   $00,X
0009-   E6 06         INC   $06
000B-   E8            INX
000C-   D0 F7         BNE   $0005

000E-   20 CF 00      JSR   $00CF

; subroutine reads a nibble and
; stores it in the accumulator
00CF-   AD EC C0      LDA   $C0EC         o_O
00D2-   10 FB         BPL   $00CF
00D4-   60            RTS
```

Continuing from $0011...

```
; first nibble must be $D5
0011-   C9 D5         CMP   #$D5
0013-   D0 F9         BNE   $000E

; read second nibble, must be $AA
0015-   20 CF 00      JSR   $00CF
0018-   C9 AA         CMP   #$AA
001A-   D0 F5         BNE   $0011
```

```
; We actually need the Y register to be
; $AA for unrelated reasons later, so
; let's set that now. (We have time,
; and it saves 1 byte!)
001C-   A8          TAY

; read the third nibble
001D-   20 CF 00    JSR   $00CF

; is it $AD?
0020-   49 AD       EOR   #$AD

; Yes, which means this is the data
; prologue. Branch forward to start
; reading the data field.
0022-   F0 1F       BEQ   $0043
```

If that third nibble is not $AD, we assume it's the end of the address prologue. ($96 would be the third nibble of a standard address prologue, but we don't actually check.) We fall through and start decoding the 4-4 encoded values in the address field.

```
0024-   A0 02       LDY   #$02
```

The first time through this loop, we'll read the disk volume number. The second time, we'll read the track number. The third time, we'll read the physical sector number. We don't actually care about the disk volume or the track number, and once we get the sector number, we don't verify the address field checksum.

```
0026-   20 CF 00     JSR   $00CF
0029-   2A           ROL
002A-   85 AC        STA   $AC
002C-   20 CF 00     JSR   $00CF
002F-   25 AC        AND   $AC
0031-   88           DEY
0032-   10 F2        BPL   $0026

; store the physical sector number
; (will re-use later)
0034-   85 AC        STA   $AC

; use physical sector number as an
; index into the sector address array
0036-   A8           TAY

; get the target page (where we want to
; store this sector in memory)
0037-   B6 F0        LDX   $F0,Y

; store the target page in several
; places throughout the following code
0039-   86 9B        STX   $9B
003B-   CA           DEX
003C-   86 6B        STX   $6B
003E-   86 83        STX   $83
0040-   E8           INX

; This is an unconditional branch,
; because the ROL at $0029 will always
; set the carry. We're done processing
; the address field, so we need to loop
; back and wait for the data prologue.
0041-   B0 CB        BCS   $000E

; execution continues here (from $0022)
; after matching the data prologue
0043-   E0 00        CPX   #$00
```

```
; If X is still $00, it means we found
; a data prologue before we found an
; address prologue. In that case, we
; have to skip this sector, because we
; don't know which sector it is and we
; wouldn't know where to put it.
0045-   F0 C7        BEQ   $000E
```

Nibble loop #1 reads nibbles $00..$55, looks up the corresponding offset in the preshift table at $0300, and stores that offset in the temporary buffer at $036A.

```
; initialize rolling checksum to $00
0047-   85 55        STA   $55
0049-   AE EC C0     LDX   $C0EC      o_
004C-   10 FB        BPL   $0049

; The nibble value is in the X register
; now. The lowest possible nibble value
; is $96 and the highest is $FF. To
; look up the offset in the table at
; $0300, we need to subtract $96 from
; $0300 and add X.
004E-   BD 6A 02     LDA   $026A,X

; Now the accumulator has the offset
; into the table of individual 2-bit
; combinations ($0200..$02FF). Store
; that offset in the temporary buffer
; at $036A, in the order we read the
; nibbles. But the Y register started
; counting at $AA, so we need to
; subtract $AA from $036A and add Y.
0051-   99 C0 02     STA   $02C0,Y
```

```
; The EOR value is set at $0047
; each time through loop #1.
0054-   49 00       EOR   #$00      /!\
0056-   C8          INY
0057-   D0 EE       BNE   $0047
```

Here endeth nibble loop #1.

Nibble loop #2 reads nibbles $56..$AB, combines them with bits 0-1 of the appropriate nibble from the first $56, and stores them in bytes $00..$55 of the target page in memory.

```
0059-   A0 AA       LDY   #$AA
005B-   AE EC C0    LDX   $C0EC     o_O
005E-   10 FB       BPL   $005B
0060-   5D 6A 02    EOR   $026A,X
0063-   BE C0 02    LDX   $02C0,Y
0066-   5D 02 02    EOR   $0202,X

; This address was set at $003C
; based on the target page (minus 1
; so we can add Y from $AA..$FF).
0069-   99 56 D1    STA   $D156,Y   /!\
006C-   C8          INY
006D-   D0 EC       BNE   $005B
```

Here endeth nibble loop #2.

Nibble loop #3 reads nibbles $AC..$101, combines them with bits 2-3 of the appropriate nibble from the first $56, and stores them in bytes $56..$AB of the target page in memory.

```
006F-    29 FC          AND   #$FC
0071-    A0 AA          LDY   #$AA
0073-    AE EC C0       LDX   $C0EC      o_O
0076-    10 FB          BPL   $0073
0078-    5D 6A 02       EOR   $026A,X
007B-    BE C0 02       LDX   $02C0,Y
007E-    5D 01 02       EOR   $0201,X

; This address was set at $003E
; based on the target page (minus 1
; so we can add Y from $AA..$FF).
0081-    99 AC D1       STA   $D1AC,Y    /!\
0084-    C8             INY
0085-    D0 EC          BNE   $0073
```

Here endeth nibble loop #3.

Loop #4 reads nibbles $102..$155, combines them with bits 4-5 of the appropriate nibble from the first $56, and stores them in bytes $AC..$FF of the target page in memory.

```
0087-    29 FC          AND   #$FC
0089-    A2 AC          LDX   #$AC
008B-    AC EC C0       LDY   $C0EC      o_O
008E-    10 FB          BPL   $008B
0090-    59 6A 02       EOR   $026A,Y
0093-    BC BE 02       LDY   $02BE,X
0096-    59 00 02       EOR   $0200,Y

; This address was set at $0039
; based on the target page.
0099-    9D 00 D1       STA   $D100,X    /!\
009C-    E8             INX
009D-    D0 EC          BNE   $008B
```

Here endeth nibble loop #4.

```
; Finally, get the last nibble,
; which is the checksum of all
; the previous nibbles.
009F-   29 FC       AND   #$FC
00A1-   AC EC C0    LDY   $C0EC     o_O
00A4-   10 FB       BPL   $00A1
00A6-   59 6A 02    EOR   $026A,Y

; if checksum fails, start over
00A9-   D0 96       BNE   $0041

; This was set to the physical
; sector number (at $0034), so
; this is a index into the 16-
; byte array at $00F0.
00AB-   A0 C0       LDY   #$C0      /!\

; store $00 at this index in the sector
; array to indicate that we've read
; this sector
00AD-   96 F0       STX   $F0,Y

; are we done yet?
00AF-   E6 00       INC   $00

; nope, loop back to read more sectors
00B1-   D0 8E       BNE   $0041

; And that's all she read.
00B3-   60          RTS
```

0boot's track read routine is done when $0000 hits $00, which is astonishingly beautiful. Like, "now I know God" level of beauty.

And so it goes: we pop another address off the stack, move the drive arm, read another track, and so on. Eventually we finish moving and reading, moving and reading, and we get to the home stretch and start calling ROM routines.

```
  $FE88 (IN#0, pushed at $0831)
  $FE92 (PR#0, pushed at $0831)
  $FE83 (NORMAL, pushed at $0831)
  $FB2E (TEXT, pushed at $0831)
```

It turns out that this game is very sensitive to the text-related zero page locations like the I/O vectors and the FLASH/INVERSE/NORMAL text mask. It uses bog-standard "JSR $FDED" to print text characters on the screen for the main menu, which fails spectacularly if zero page still has executable code in it. Thus, reset everything, and life is good.

Next on the stack:

```
  $00D4

; turn off drive motor
00D5-   AD E8 C0    LDA   $C0E8     /!\
```

And the last thing on the stack:

```
  $07FF
```

...which jumps to $0800 and starts the game.

Quod erat liberandum.

Acknowledgements

Thanks to LoGo for the original disk. I've been looking for this one for years.

Thanks to qkumba for writing 0boot, for explaining 6-and-2 encoding to me, and for unfailingly being that rare combination of smart and kind.


```
----------------------------------------
A 4am crack                      No. 673
------------------EOF-------------------
```