Outpost

2015-12-12

Contents

-----------------Outpost------------------
A 4am crack   2015-12-12

Name: Outpost
Genre: arcade
Year: 1981
Authors: Tom McWilliams
Publisher: Sirius Software
Media: single-sided 5.25-inch floppy
OS: custom
Previous cracks: two different file
  cracks, both uncredited
Similar cracks:
  #315 Beer Run

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)
hangs during boot

Copy][+ nibble editor
track 0 has some 4-4 encoded data
other tracks are unreadable

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?
I don't know. Could be a nibble check during boot. Could be that the data is loaded from half tracks. Could be both, or neither.

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 Find A Very Unfriendly "Do Not Disturb" Sign

```
[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

; display hi-res graphics page
; (uninitialized)
0801-   8D 50 C0     STA   $C050
0804-   8D 52 C0     STA   $C052
0807-   8D 54 C0     STA   $C054
080A-   8D 57 C0     STA   $C057

; get slot (x16)
080D-   A6 2B        LDX   $2B

; a counter? or an address?
080F-   A9 04        LDA   #$04
0811-   85 11        STA   $11
0813-   A0 00        LDY   #$00
0815-   84 10        STY   $10
```

```
; look for custom prologue ("DD AD DA")
0817-   BD 8C C0    LDA   $C08C,X
081A-   10 FB       BPL   $0817
081C-   C9 DD       CMP   #$DD
081E-   D0 F7       BNE   $0817
0820-   BD 8C C0    LDA   $C08C,X
0823-   10 FB       BPL   $0820
0825-   C9 AD       CMP   #$AD
0827-   D0 F3       BNE   $081C
0829-   BD 8C C0    LDA   $C08C,X
082C-   10 FB       BPL   $0829
082E-   C9 DA       CMP   #$DA
0830-   D0 EA       BNE   $081C

; read 4-4 encoded data immediately
; (no address field, no sector numbers)
0832-   BD 8C C0    LDA   $C08C,X
0835-   10 FB       BPL   $0832
0837-   38          SEC
0838-   2A          ROL
0839-   85 0E       STA   $0E
083B-   BD 8C C0    LDA   $C08C,X
083E-   10 FB       BPL   $083B
0840-   25 0E       AND   $0E

; ($10) is an address, initialized at
; $080F as $0400 (yes, the text page)
0842-   91 10       STA   ($10),Y
0844-   C8          INY
0845-   D0 EB       BNE   $0832
0847-   E6 11       INC   $11
0849-   A5 11       LDA   $11

; loop until we hit page 8 (i.e. we're
; filling $0400..$07FF)
084B-   C9 08       CMP   #$08
084D-   D0 E3       BNE   $0832
084F-   BD 80 C0    LDA   $C080,X
```

```
; clear $0900..$BFFF in main memory
0852-   A9 09         LDA   #$09
0854-   85 01         STA   $01
0856-   A9 00         LDA   #$00
0858-   85 00         STA   $00
085A-   A8            TAY
085B-   A2 B7         LDX   #$B7
085D-   91 00         STA   ($00),Y
085F-   C8            INY
0860-   D0 FB         BNE   $085D
0862-   E6 01         INC   $01
0864-   CA            DEX
0865-   D0 F6         BNE   $085D

; calculate a checksum of page 8 (this
; code right here)
0867-   8A            TXA
0868-   E8            INX
0869-   F0 06         BEQ   $0871
086B-   5D 00 08      EOR   $0800,X
086E-   4C 68 08      JMP   $0868

; use the stack pointer (!) to keep a
; copy of that checksum
0871-   AA            TAX
0872-   9A            TXS

; calculate another checksum of zero
; page
0873-   A2 00         LDX   #$00
0875-   8A            TXA
0876-   55 00         EOR   $00,X
0878-   E8            INX
0879-   D0 FB         BNE   $0876

; get slot (x16) again
087B-   A6 2B         LDX   $2B
```

```
; jump to the code we just read into
; the text page
087D-   4C 00 04    JMP   $0400
```

Well that's lovely. I need to interrupt the boot at $087D, but if I do, it will modify the checksum that ends up in the stack pointer (which is a great place to stash a checksum as long as you never use PHA, PLA, PHP, PLP, JSR, RTS, or RTI).

It's also wiping main memory, including the place I usually put my boot trace callbacks (around $9700).

So, a three-pronged attack:

1. Relocate the code to $0900. Most of it uses relative branching already, except for one JMP at $086E, which I can patch. The code will still run, but I'll be able to patch it without altering the checksum.
2. Disable the memory wipe at $095D.
3. Patch the code at $097D to jump to a routine under my control.

Chapter 2
In Which Nothing Happens, Inhospitably

```
*9600<C600.C6FFM

; relocate the code from $0800 to $0900
96F8-   A0 00       LDY   #$00
96FA-   B9 00 08    LDA   $0800,Y
96FD-   99 00 09    STA   $0900,Y
9700-   C8          INY
9701-   D0 F7       BNE   $96FA

; disable the memory wipe by changing
; STA to BIT
9703-   A9 24       LDA   #$24
9705-   8D 5D 09    STA   $095D

; fix the absolute JMP address
9708-   A9 09       LDA   #$09
970A-   8D 70 09    STA   $0970

; set up the callback
970D-   A9 1A       LDA   #$1A
970F-   8D 7E 09    STA   $097E
9712-   A9 97       LDA   #$97
9714-   8D 7F 09    STA   $097F

; start the boot
9717-   4C 01 09    JMP   $0901
```

```
; callback is here
; copy the code on the text page to
; higher memory so it will survive a
; reboot
971A-   A2 04           LDX    #$04
971C-   A0 00           LDY    #$00
971E-   B9 00 04        LDA    $0400,Y
9721-   99 00 24        STA    $2400,Y
9724-   C8              INY
9725-   D0 F7           BNE    $971E
9727-   EE 20 97        INC    $9720
972A-   EE 23 97        INC    $9723
972D-   CA              DEX
972E-   D0 EE           BNE    $971E

; turn off slot 6 drive motor and
; reboot to my work disk in slot 5
9730-   AD E8 C0        LDA    $C0E8
9733-   4C 00 C5        JMP    $C500

*BSAVE TRACE,A$9600,L$136
*9600G
...reboots slot 6...
...reboots slot 5...

]BSAVE BOOT1 0400-07FF,A$2400,L$400
]CALL -151
```

I'm going to leave this code at $2400. Relative branches will look correct, but absolute addresses will be off by $2000.

```
*2400L

; calculate another checksum of zero
; page, starting with the value of the
; previous checksum (at $0873)
2400-    A0 00       LDY   #$00
2402-    59 00 00    EOR   $0000,Y
2405-    C8          INY
2406-    D0 FA       BNE   $2402
2408-    A8          TAY

; if equal, nothing has changed (we've
; EOR'd everything twice, so we're back
; to zero)
2409-    F0 03       BEQ   $240E

; if checksums don't match, jump to
; (what I presume is) The Badlands
240B-    4C 74 07    JMP   $0774

*2774L

; several entry points (also at $0770)
; for errors at different points in the
; boot process
2774-    A9 01       LDA   #$01
2776-    D0 10       BNE   $2788
2778-    A9 02       LDA   #$02
277A-    D0 0C       BNE   $2788
277C-    A9 03       LDA   #$03
277E-    D0 08       BNE   $2788
2780-    A9 04       LDA   #$04
2782-    D0 04       BNE   $2788
2784-    A9 05       LDA   #$05
2786-    D0 00       BNE   $2788
2788-    09 B0       ORA   #$B0
278A-    2C 00 08    BIT   $0800
278D-    20 2F FB    JSR   $FB2F
2790-    AD 55 C0    LDA   $C055
```

```
; clear the rest of main memory,
; starting at $0801
2793-   A9 08       LDA   #$08
2795-   85 01       STA   $01
2797-   A9 01       LDA   #$01
2799-   85 00       STA   $00
279B-   A0 00       LDY   #$00
279D-   A2 B8       LDX   #$B8
279F-   A9 A0       LDA   #$A0
27A1-   91 00       STA   ($00),Y
27A3-   C8          INY
27A4-   D0 FB       BNE   $27A1
27A6-   E6 01       INC   $01
27A8-   CA          DEX
27A9-   D0 F6       BNE   $27A1

; play a cute sound
27AB-   A9 08       LDA   #$08
27AD-   85 00       STA   $00
27AF-   A0 80       LDY   #$80
27B1-   AD 30 C0    LDA   $C030
27B4-   A2 60       LDX   #$60
27B6-   CA          DEX
27B7-   D0 FD       BNE   $27B6
27B9-   88          DEY
27BA-   D0 F5       BNE   $27B1
27BC-   C6 00       DEC   $00
27BE-   D0 EF       BNE   $27AF
```

```
; and reboot from whence we came
27C0-   A6 2B       LDX   $2B
27C2-   CA          DEX
27C3-   8A          TXA
27C4-   4A          LSR
27C5-   4A          LSR
27C6-   4A          LSR
27C7-   4A          LSR
27C8-   09 C0       ORA   #$C0
27CA-   48          PHA
27CB-   A9 FF       LDA   #$FF
27CD-   48          PHA
27CE-   60          RTS
```

Continuing from $040E...

```
*240EL

; set reset vector to The Badlands
240E-   A9 70       LDA   #$70
2410-   8D F2 03    STA   $03F2
2413-   A9 07       LDA   #$07
2415-   8D F3 03    STA   $03F3
2418-   49 A5       EOR   #$A5
241A-   8D F4 03    STA   $03F4
241D-   86 2B       STX   $2B
241F-   EA          NOP

; read from ROM but write to RAM bank 2
2420-   AD 81 C0    LDA   $C081
2423-   AD 81 C0    LDA   $C081
```

```
; wipe RAM bank 2 by copying ROM
2426-   A0 00       LDY   #$00
2428-   84 00       STY   $00
242A-   A9 D0       LDA   #$D0
242C-   85 01       STA   $01
242E-   B1 00       LDA   ($00),Y
2430-   91 00       STA   ($00),Y
2432-   C8          INY
2433-   D0 F9       BNE   $242E
2435-   E6 01       INC   $01
2437-   D0 F5       BNE   $242E

; set low-level reset vector while the
; language card RAM is writeable (also
; to The Badlands)
2439-   A9 70       LDA   #$70
243B-   8D FC FF    STA   $FFFC
243E-   A9 07       LDA   #$07
2440-   8D FD FF    STA   $FFFD

; switch back to ROM
2443-   AD 80 C0    LDA   $C080

; set input and output vectors to
; The Badlands
2446-   A9 74       LDA   #$74
2448-   85 36       STA   $36
244A-   85 38       STA   $38
244C-   A9 07       LDA   #$07
244E-   85 37       STA   $37
2450-   85 39       STA   $39
```

```
; take the checksum from boot0 (that we
; stashed in the stack pointer) and put
; it in zero page $0B
2452-   A9 00       LDA   #$00
2454-   BA          TSX
2455-   86 0B       STX   $0B
2457-   85 0C       STA   $0C
2459-   85 0D       STA   $0D
245B-   85 0E       STA   $0E

; use that checksum (now in zero page
; $0B) as the starting value of ANOTHER
; checksum of all the code on the text
; page (including this code right here)
245D-   A5 0B       LDA   $0B
245F-   A2 00       LDX   #$00
2461-   5D 00 04    EOR   $0400,X
2464-   5D 00 05    EOR   $0500,X
2467-   5D 00 06    EOR   $0600,X
246A-   5D 00 07    EOR   $0700,X
246D-   E8          INX
246E-   D0 F1       BNE   $2461

; push the new checksum to the stack,
; twice
2470-   48          PHA
2471-   48          PHA
```

Chapter 3
You're Very Clever, Young Man, But It's Checksums All The Way Down

```
*9600<C600.C6FFM

; move boot0 to $0900 and patch it up
96F8-    A0 00          LDY   #$00
96FA-    B9 00 08       LDA   $0800,Y
96FD-    99 00 09       STA   $0900,Y
9700-    C8             INY
9701-    D0 F7          BNE   $96FA
9703-    A9 24          LDA   #$24
9705-    8D 5D 09       STA   $095D
9708-    A9 09          LDA   #$09
970A-    8D 70 09       STA   $0970

; set up callback after first checksum
; is calculated
970D-    A9 1A          LDA   #$1A
970F-    8D 7E 09       STA   $097E
9712-    A9 97          LDA   #$97
9714-    8D 7F 09       STA   $097F

; start the boot
9717-    4C 01 09       JMP   $0901

; callback is here
; save the checksum and unconditionally
; break to the monitor
971A-    BA             TSX
971B-    8A             TXA
971C-    8D FF 97       STA   $97FF
971F-    AD E8 C0       LDA   $C0E8
9722-    4C 59 FF       JMP   $FF59

*BSAVE TRACE 0872 CHECKSUM,A$9600,L$125
*9600G
...reboots slot 6...
<beep>
```

```
*97FF

97FF- 20
```

The initial checksum of boot0 is $20.

```
*C500G
...
]CALL -151

*9600<C600.C6FFM

; move boot0 to $0900 and patch it up
96F8-    A0 00          LDY   #$00
96FA-    B9 00 08       LDA   $0800,Y
96FD-    99 00 09       STA   $0900,Y
9700-    C8             INY
9701-    D0 F7          BNE   $96FA
9703-    A9 24          LDA   #$24
9705-    8D 5D 09       STA   $095D
9708-    A9 09          LDA   #$09
970A-    8D 70 09       STA   $0970

; set up callback instead of jumping to
; boot1 at $0400
970D-    A9 1A          LDA   #$1A
970F-    8D 7E 09       STA   $097E
9712-    A9 97          LDA   #$97
9714-    8D 7F 09       STA   $097F

; start the boot
9717-    4C 01 09       JMP   $0901
```

```
; callback is here
; hard-code the initial checksum value
; ($20), then reproduce the checksum on
; the boot1 code before we start
; patching it to high heaven
971A-   A2 20       LDX   #$20
971C-   A9 00       LDA   #$00
971E-   86 0B       STX   $0B
9720-   85 0C       STA   $0C
9722-   85 0D       STA   $0D
9724-   85 0E       STA   $0E
9726-   A5 0B       LDA   $0B
9728-   A2 00       LDX   #$00
972A-   5D 00 04    EOR   $0400,X
972D-   5D 00 05    EOR   $0500,X
9730-   5D 00 06    EOR   $0600,X
9733-   5D 00 07    EOR   $0700,X
9736-   E8          INX
9737-   D0 F1       BNE   $972A

; store the new checksum and break
9739-   8D FF 97    STA   $97FF
973C-   AD E8 C0    LDA   $C0E8
973F-   4C 59 FF    JMP   $FF59

*BSAVE TRACE 0470 CHECKSUM,A$960,L$142
*9600G
...reboots slot 6...
<beep>

*97FF

97FF- 00
```

The second checksum, which gets pushed twice to the stack at $0470, is $00.

Chapter 4
In Which Half A Track Is Better Than None

Continuing the boot trace at $0472...

```
*C500G
...
]BLOAD BOOT1 0400-07FF,A$2400
]CALL -151

*2472L

2472-   A0 03       LDY   #$03
2474-   20 00 05    JSR   $0500

*2500L

2500-   20 DC 04    JSR   $04DC

*24DCL

; advance drive head by one phase
; (a.k.a. a half track)
24DC-   E6 0C       INC   $0C
24DE-   A5 0C       LDA   $0C
24E0-   29 03       AND   #$03
24E2-   0A          ASL
24E3-   05 2B       ORA   $2B
24E5-   AA          TAX
24E6-   BD 81 C0    LDA   $C081,X
24E9-   20 F8 04    JSR   $04F8
24EC-   BD 80 C0    LDA   $C080,X
24EF-   20 F8 04    JSR   $04F8
```

```
; loop a number of times (given in the
; Y register on entry)
24F2-   88          DEY
24F3-   D0 E7       BNE   $24DC
24F5-   A6 2B       LDX   $2B
24F7-   60          RTS
24F8-   8D 50 C0    STA   $C050
24FB-   A9 40       LDA   #$40
24FD-   4C A8 FC    JMP   $FCA8
```

We started on track 0 and advanced the drive head by 3 phases, so now we're on track 1.5.

Continuing from $0503...

```
; save X, display "Outpost" on hi-res
; screen (not shown), restore X
2503-   86 2B       STX   $2B
2505-   A9 00       LDA   #$00
2507-   A2 0F       LDX   #$0F
2509-   A0 18       LDY   #$18
250B-   20 60 05    JSR   $0560
250E-   A6 2B       LDX   $2B
2510-   60          RTS
```

Continuing from $0477...

```
*2477L

; get target memory page from an array
; at $05F0
2477-   A4 0E       LDY   $0E
2479-   B9 F0 05    LDA   $05F0,Y

; when page = 0, jump to next stage
; at $0520, otherwise continue at $0481
247C-   D0 03       BNE   $2481
247E-   4C 20 05    JMP   $0520
2481-   20 90 04    JSR   $0490

*2490L

; sector count (4-4 encoded tracks can
; only hold $0C pages worth of data)
2490-   85 05       STA   $05
2492-   18          CLC
2493-   A9 0C       LDA   #$0C
2495-   85 06       STA   $06
2497-   A0 00       LDY   #$00
2499-   84 04       STY   $04

; custom prologue "DD AD DA"
249B-   BD 8C C0    LDA   $C08C,X
249E-   10 FB       BPL   $249B
24A0-   C9 DD       CMP   #$DD
24A2-   D0 F7       BNE   $249B
24A4-   BD 8C C0    LDA   $C08C,X
24A7-   10 FB       BPL   $24A4
24A9-   C9 AD       CMP   #$AD
24AB-   D0 F3       BNE   $24A0
24AD-   BD 8C C0    LDA   $C08C,X
24B0-   10 FB       BPL   $24AD
24B2-   C9 DA       CMP   #$DA
24B4-   D0 EA       BNE   $24A0
```

```
; now read 4-4 encoded data into ($04)
24B6-   BD 8C C0     LDA   $C08C,X
24B9-   10 FB        BPL   $24B6
24BB-   8D 57 C0     STA   $C057
24BE-   38           SEC
24BF-   2A           ROL
24C0-   8D 50 C0     STA   $C050
24C3-   85 0F        STA   $0F
24C5-   BD 8C C0     LDA   $C08C,X
24C8-   10 FB        BPL   $24C5
24CA-   25 0F        AND   $0F
24CC-   91 04        STA   ($04),Y
24CE-   C8           INY
24CF-   D0 E5        BNE   $24B6

; increment target page
24D1-   E6 05        INC   $05

; decrement count
24D3-   C6 06        DEC   $06

; Loop back to read more. Note: this
; goes directly to data read routine,
; not the prologue match routine. There
; is only one prologue per track.
24D5-   D0 DF        BNE   $24B6
24D7-   60           RTS
```

Continuing from $0484...

```
*2484L

; sets Y=2 and falls through to drive
; head advance routine, so this will
; skip ahead 2 phases = 1 whole track,
; so we're still on half tracks but now
; 2.5, 3.5, 4.5, &c.
2484-   20 D8 04    JSR   $04D8

; show hi-res screen, increment index
; into page array, and jump back to
; read the next track
2487-   8D 50 C0    STA   $C050
248A-   E6 0E       INC   $0E
248C-   4C 77 04    JMP   $0477
```

Here is the target page table (accessed at $0479):

```
*25F0.

25F0- 08 14 40 4C 58 64 70 7C
25F8- 88 00
```

Each call to $0490 reads $0C sectors, so we're filling $0800..$1FFF, skipping hi-res screen 1 (initialized earlier with the graphical "Outpost" loading screen), then filling $4000..$93FF.

Once the page array is exhausted, $047E jumps to $0520 for the next boot stage.

To sum up:

- We're reading data from consecutive half tracks (1.5, 2.5, 3.5, &c.)
- Each track has $0C pages of data in a custom (non-sector-based) format
- We're using $0800..$93FF in main memory (hi-res screen 1 was drawn earlier, then the rest is read directly from disk)
- Nothing in this read loop relies on the checksum we stashed in the stack pointer or the later checksum we pushed twice to the stack
- $047E exits via $0520

Let's capture it.

Chapter 5
In Which Things Have Been Made As Difficult As Possible For Us

```
*9600<C600.C6FFM

*96F8L

; move boot0 to $0900 and patch it up
96F8-    A0 00          LDY   #$00
96FA-    B9 00 08       LDA   $0800,Y
96FD-    99 00 09       STA   $0900,Y
9700-    C8             INY
9701-    D0 F7          BNE   $96FA
9703-    A9 24          LDA   #$24
9705-    8D 5D 09       STA   $095D
9708-    A9 09          LDA   #$09
970A-    8D 70 09       STA   $0970

; set up callback before jumping to
; $0400
970D-    A9 1A          LDA   #$1A
970F-    8D 7E 09       STA   $097E
9712-    A9 97          LDA   #$97
9714-    8D 7F 09       STA   $097F

; start the boot
9717-    4C 01 09       JMP   $0901

; initialize zero page (copied verbatim
; from $0457)
971A-    A9 00          LDA   #$00
971C-    85 0B          STA   $0B
971E-    85 0C          STA   $0C
9720-    85 0D          STA   $0D
9722-    85 0E          STA   $0E
```

```
; break to the monitor at $047E instead
; of continuing at $0520
9724-   A9 4C        LDA   #$4C
9726-   8D 7E 04     STA   $047E
9729-   A9 59        LDA   #$59
972B-   8D 7F 04     STA   $047F
972E-   A9 FF        LDA   #$FF
9730-   8D 80 04     STA   $0480
9733-   4C 72 04     JMP   $0472

*BSAVE TRACE2,A$9600,L$136

; fill main memory so I can verify
; which pages changed (in case I made
; a mistake in my analysis earlier!)
*800:0 N 801<800.BEFEM

*BRUN TRACE2
...reboots slot 6...
<beep>
```

A quick inspection of memory confirms that $0800..$93FF have changed, and the rest are untouched (except the text page, but I knew that).

According to "Inside the Apple //e" (pp. 296-8), $C311 copies data from main memory to aux memory and back. (Aux memory is what you get by having an 80-column card, 128K instead of 64.)

```
The routine itself takes 4 parameters:

  ($3C/$3D) starting address
  ($3E/$3F) ending address
  ($42/$43) destination address in the
            other memory bank
  carry bit set for main->aux copy, or
            clear for aux->main copy

Thus, to copy $0800..$93FF to auxiliary
memory:

0300-   A9 00           LDA   #$00
0302-   85 3C           STA   $3C
0304-   85 42           STA   $42
0306-   A9 08           LDA   #$08
0308-   85 3D           STA   $3D
030A-   85 43           STA   $43
030C-   A9 FF           LDA   #$FF
030E-   85 3E           STA   $3E
0310-   A9 93           LDA   #$93
0312-   85 3F           STA   $3F
0314-   38              SEC
0315-   4C 11 C3        JMP   $C311

*300G

; reboot to my work disk
*C500G
...
]CALL -151
```

And copy $0800..$93FF from auxiliary memory back to main memory, I only need to change the "SEC" to "CLC" at $0314:

```
0300-   A9 00       LDA   #$00
0302-   85 3C       STA   $3C
0304-   85 42       STA   $42
0306-   A9 08       LDA   #$08
0308-   85 3D       STA   $3D
030A-   85 43       STA   $43
030C-   A9 FF       LDA   #$FF
030E-   85 3E       STA   $3E
0310-   A9 93       LDA   #$93
0312-   85 3F       STA   $3F
0314-   18          CLC
0315-   4C 11 C3    JMP   $C311

*300G

*BSAVE OBJ,A$800,L$8C00
```

Continuing from $0520...

```
*BLOAD BOOT1 0400-07FF,A$2400
*2520L

; turn off drive motor
2520-   BD 88 C0    LDA   $C088,X
2523-   20 D0 07    JSR   $07D0
```

```
*27D0L

; calculate a simple one-byte checksum
; on the entire game code (minus hi-res
; graphics screen 1) to ensure the game
; code has not been tampered with
27D0-   A0 00       LDY   #$00
27D2-   84 04       STY   $04
27D4-   A9 08       LDA   #$08
27D6-   85 05       STA   $05
27D8-   A9 00       LDA   #$00
27DA-   51 04       EOR   ($04),Y
27DC-   C8          INY
27DD-   D0 FB       BNE   $27DA
27DF-   E6 05       INC   $05
27E1-   A6 05       LDX   $05

; skip from $2000 to $4000
27E3-   E0 20       CPX   #$20
27E5-   D0 F3       BNE   $27DA
27E7-   06 05       ASL   $05

; continue calculating checksum in the
; accumulator
27E9-   51 04       EOR   ($04),Y
27EB-   C8          INY
27EC-   D0 FB       BNE   $27E9
27EE-   E6 05       INC   $05
27F0-   A6 05       LDX   $05
27F2-   E0 94       CPX   #$94
27F4-   D0 F3       BNE   $27E9
27F6-   A8          TAY
```

```
; if checksum fails, it's off to The
; Badlands with you!
27F7-   D0 87        BNE   $2780
27F9-   60           RTS
```

Continuing from $0526...

```
*2526L

; get those checksum values we pushed
; to the stack at $0470 and start
; fiddling with them
2526-   68           PLA         ;A=$00
2527-   AA           TAX         ;X=$00
2528-   68           PLA         ;A=$00
2529-   38           SEC
252A-   69 7E        ADC   #$7E  ;A=$7F
252C-   48           PHA         ;S+$7F
252D-   8A           TXA         ;A=$00
252E-   18           CLC
252F-   E9 00        SBC   #$00
2531-   48           PHA         ;S+$FF
2532-   38           SEC
2533-   69 36        ADC   #$36  ;A=$36
2535-   85 00        STA   $00
2537-   38           SEC
2538-   E9 36        SBC   #$36  ;A=$00
253A-   85 01        STA   $01
```

($00) points to $0036 now.

```
253C-   A8           TAY         ;Y=$00
253D-   68           PLA         ;A=$FF
253E-   48           PHA
```

Still $7F/$FF on the stack.

```
253F-   18          CLC
2540-   69 64       ADC   #$64    ;A=$63
2542-   91 00       STA   ($00),Y
```

zp$36 = $63 now.

```
2544-   C8          INY           ;Y=$01
2545-   38          SEC
2546-   69 00       ADC   #$00    ;A=$64
2548-   91 00       STA   ($00),Y
```

zp$37 = $64 now.

```
254A-   A9 00       LDA   #$00
254C-   85 00       STA   $00
```

($00) points to $0000 now.

```
254E-   68          PLA           ;A=$FF
254F-   48          PHA
```

Still $7F/$FF on the stack.

```
2550-   91 00       STA   ($00),Y
```

zp$00 = $FF now. ($00) points to $00FF.

```
2552-   C8          INY           ;Y=$01
2553-   38          SEC
2554-   E9 08       SBC   #$08    ;A=$F7
2556-   91 00       STA   ($00),Y
```

$0100 = $F7 now.

($36) points to $6463.
$0100 = $F7.
The game starts at $8000.

If I reproduce the initializations from this obfuscated routine at $0520, I should be able to run the game from the monitor. I need to do this all at once, since returning to the monitor will reset $36 and possibly $100 as well.

```
*36:63 64 N 100:F7 N 8000G
...crashes...
```

I'm missing something. Maybe a callback to the RWTS on the text page? I've seen other Sirius games do that.

```
*C500G
...
]CALL -151
*BLOAD OBJ
*BLOAD BOOT1 0400-07FF,A$9400
*36:63 64 N 100:F7 N 400<9400.97FFM N
 8000G
...crashes...
```

Still no luck. Maybe some secondary protection in the game code? Or even a secondary loader? (I've seen both in other Sirius games.)

Sigh. Let's start tracing through the code at $8000.

Chapter 6
And One More Thing

```
*BLOAD OBJ
*8000L

8000-   20 00 81    JSR   $8100

*8100L

; harmless
8100-   A9 C0       LDA   #$C0
8102-   85 45       STA   $45
8104-   A9 00       LDA   #$00
8106-   85 58       STA   $58
8108-   85 83       STA   $83
810A-   85 84       STA   $84
810C-   85 85       STA   $85
810E-   85 86       STA   $86
8110-   85 87       STA   $87
8112-   85 88       STA   $88
8114-   A9 01       LDA   #$01
8116-   85 8B       STA   $8B
8118-   85 60       STA   $60
811A-   4C 30 81    JMP   $8130

*8130L

; harmless
8130-   8D 3F 76    STA   $763F
8133-   A9 00       LDA   #$00
8135-   85 80       STA   $80
8137-   85 81       STA   $81
8139-   85 82       STA   $82
813B-   85 8C       STA   $8C
813D-   4C C0 8E    JMP   $8EC0
```

```
*8EC0L

; hmm
8EC0-    20 00 8F     JSR   $8F00
8EC3-    B0 0B        BCS   $8ED0
8EC5-    20 0B 8F     JSR   $8F0B
8EC8-    B0 06        BCS   $8ED0
8ECA-    4C 70 07     JMP   $0770
8ECD-    00           BRK
8ECE-    00           BRK
8ECF-    00           BRK
8ED0-    BD 88 C0     LDA   $C088,X
8ED3-    60           RTS

*8F00L

; turn on boot slot drive motor
; (DEFINITELY NOT HARMLESS)
8F00-    A6 2B        LDX   $2B
8F02-    BD 89 C0     LDA   $C089,X

; advance drive by 2 phases (=1 track)
8F05-    A0 02        LDY   #$02
8F07-    20 DC 04     JSR   $04DC
8F0A-    EA           NOP
8F0B-    A9 00        LDA   #$00
8F0D-    85 01        STA   $01
8F0F-    A8           TAY
```

```
; look for prologue, "D5 AA AD"
8F10-   BD 8C C0      LDA   $C08C,X
8F13-   10 FB         BPL   $8F10
8F15-   C9 D5         CMP   #$D5
8F17-   D0 F7         BNE   $8F10
8F19-   BD 8C C0      LDA   $C08C,X
8F1C-   10 FB         BPL   $8F19
8F1E-   C9 AA         CMP   #$AA
8F20-   D0 F3         BNE   $8F15
8F22-   BD 8C C0      LDA   $C08C,X
8F25-   10 FB         BPL   $8F22
8F27-   C9 AD         CMP   #$AD
8F29-   D0 EA         BNE   $8F15

; count nibbles until epilogue, "DE AA"
8F2B-   C8            INY
8F2C-   D0 04         BNE   $8F32
8F2E-   E6 01         INC   $01
8F30-   F0 17         BEQ   $8F49
8F32-   BD 8C C0      LDA   $C08C,X
8F35-   10 F9         BPL   $8F30
8F37-   C9 DE         CMP   #$DE
8F39-   D0 F0         BNE   $8F2B
8F3B-   BD 8C C0      LDA   $C08C,X
8F3E-   10 F9         BPL   $8F39
8F40-   C9 AA         CMP   #$AA
8F42-   D0 E7         BNE   $8F2B

; if >= $0C00 nibbles between prologue
; and epilogue, carry is set on exit
8F44-   A5 01         LDA   $01
8F46-   C9 0C         CMP   #$0C
8F48-   60            RTS
8F49-   38            SEC
8F4A-   60            RTS
```

Returning to $8EC0...

```
*8EC0L

; count nibbles
8EC0-   20 00 8F    JSR   $8F00

; carry set = success, exit via $8ED0
8EC3-   B0 0B       BCS   $8ED0

; count nibbles again (but stay on the
; same track)
8EC5-   20 0B 8F    JSR   $8F0B

; carry set = success, exit via $8ED0
8EC8-   B0 06       BCS   $8ED0

; failure --> The Badlands
8ECA-   4C 70 07    JMP   $0770
8ECD-   00          BRK
8ECE-   00          BRK
8ECF-   00          BRK

; success path, turn off drive motor
; and return gracefully
8ED0-   BD 88 C0    LDA   $C088,X
8ED3-   60          RTS
```

I should be able to put an "RTS" at $8EC0 to disable this secondary protection altogether.

```
*BLOAD BOOT1 0400-07FF,A$9400
```

```
; set $36/$37, $0100, copy RWTS to text
; page, disable secondary protection,
; and jump to the game entry point
*36:63 64 N 100:F7 N 400<9400.97FFM N
 8EC0:60 N 8000G
...game works, and it is glorious...
```

I didn't bother setting zp$01 because the secondary protection overwrites it. The game doesn't appear to care about zp$01 is after it's checked at $8F46. It also doesn't seem to care about $00. However, I tried not setting $36/$37 and the graphics glitched out, so that vector is being used for something.

I might be able to reduce this further, but $0400..$93FF is a nice round number (exactly 9 tracks on a 16-sector disk), so let's move on.

Chapter 7
In Which We Step, Ever So Gently, Into The 21st Century

To reproduce the original disk's boot experience as faithfully as possible, I decided against releasing this as a file crack. The original disk displays the graphical title screen during boot. In fact, it *only* displays it during boot, then never again. Classic cracks often didn't include the title screen, because it was the 80s and 8192 bytes was expensive. The social mores of the classic crackers allowed for discarding title screens altogether in pursuit of the smallest possible file crack.

I have all the game code. I know how to initialize it and call it. Now to write it all to disk. (We'll worry about reading it back in just a minute.)

```
[S6,D1=blank formatted disk]
[S5,D1=my work disk]

]PR#5
...
]CALL -151

; page count (decremented)
0300-   A9 90       LDA   #$90
0302-   85 FF       STA   $FF

; logical sector (incremented)
0304-   A9 00       LDA   #$00
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 $90 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 $01,
; sector $00, address $1400, and RWTS
; write command ($02)
0388- 01 60 01 00 01 00 FB F7
0390- 00 14 00 00 02 00 00 60

*BSAVE MAKE,A$300,L$98
```

```
*BLOAD BOOT1 0400-07FF,A$1400
*BLOAD OBJ,A$1800

*300G          ; write game to disk
```

Now I have the entire game on tracks
$01-$09 of a standard 16-sector disk.
To read it back as quickly as possible,
I'll use qkumba's "0boot" bootloader.

Chapter 8
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
```

```
; 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 06        LDX   #$06
082F-   B5 EF        LDA   $EF,X
0831-   48           PHA
0832-   CA           DEX
0833-   D0 FA        BNE   $082F

; number of tracks to load (x2) (game-
; specific -- this game uses 9 tracks)
0835-   A0 12        LDY   #$12

; loop starts here
083F-   8A           TXA

; every other time through this loop,
; we will end up taking this branch
0840-   90 03        BCC   $0845

; 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)
0842-   48           PHA
0843-   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.
0844-   C9 38       CMP   #$38

; Push $00B3 to the stack, to "return"
; to $00B4. This routine advances the
; drive head to the next half track.
0846-   48          PHA
0847-   A9 B3       LDA   #$B3
0849-   48          PHA

; loop until done
084A-   88          DEY
084B-   D0 F2       BNE   $083F
```

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 /
  $00D4 turn off drive, disable
        secondary protection
  $0525 game-specific entry point (pops
        next two values off the stack,
        sets up zero page, and pushes
        actual game entry point)
  $0000 boot1 checksum value (twice)
--bottom--
```

Boot1 reads the game into memory from tracks $01-$09, 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 9
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 10
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
084D-   A2 40       LDX   #$40
084F-   A4 55       LDY   $55
0851-   98          TYA
0852-   0A          ASL
0853-   24 55       BIT   $55
0855-   F0 12       BEQ   $0869
0857-   05 55       ORA   $55
0859-   49 FF       EOR   #$FF
085B-   29 7E       AND   #$7E
085D-   B0 0A       BCS   $0869
085F-   4A          LSR
0860-   D0 FB       BNE   $085D
0862-   CA          DEX
0863-   8A          TXA
0864-   0A          ASL
0865-   0A          ASL
0866-   99 EA 02    STA   $02EA,Y
0869-   C6 55       DEC   $55
086B-   D0 E2       BNE   $084F
```

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
086D-   46 B7       LSR   $B7
086F-   46 B7       LSR   $B7
0871-   B5 FC       LDA   $FC,X
0873-   99 FF 01    STA   $01FF,Y
0876-   E6 AC       INC   $AC
0878-   A5 AC       LDA   $AC
087A-   25 B7       AND   $B7
087C-   D0 05       BNE   $0883
087E-   E8          INX
087F-   8A          TXA
0880-   29 03       AND   #$03
0882-   AA          TAX
0883-   C8          INY
0884-   C8          INY
0885-   C8          INY
0886-   C8          INY
0887-   C0 04       CPY   #$04
0889-   B0 E6       BCS   $0871
088B-   C8          INY
088C-   C0 04       CPY   #$04
088E-   90 DD       BCC   $086D
```

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 briefly displayed an uninitialized hi-res graphics page (originally at $0801 -- literally the first thing it does on boot). So I want to do the same. It won't be absolutely first thing, but it'll be close.

```
0890-   2C 54 C0    BIT   $C054
0893-   2C 52 C0    BIT   $C052
0896-   2C 57 C0    BIT   $C057
0899-   2C 50 C0    BIT   $C050
089C-   60          RTS
```

[Note to future self: $0890..$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 $0890.]

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 11
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 $0400)
0005-   A9 04       LDA   #$04       /!\
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 eventually pop off the final routine at $00D5:

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

; disable secondary protection
00D8-    A9 60          LDA   #$60
00DA-    8D C0 8E       STA   $8EC0
00DD-    60             RTS
```

The "RTS" at $DD will pop the next address off the stack ($05/$25) and continue at $0526. As we saw earlier, that routine immediately pops the next two values off the stack and uses them to set up $00/$01, $36/$37, $0100, and jump to the game's entry point ($8000). But we don't need to do any of that ourselves. We just need to prepare the stack, then the original code can do what it's designed to do.

Minus the protect-y bits, of course.

Quod erat liberandum.